Pandas

- It is the **most used** library for data analysis.
- It introduces **DataFrames** to deal with data.

Installing Pandas

To install pandas we simply need to type

In [1]:
```
! pip install pandas
```

```
Requirement already satisfied: pandas in c:\users\sergi\anaconda3\lib\site-package
s (1.4.4)
Requirement already satisfied: python-dateutil>=2.8.1 in c:\users\sergi\anaconda3
\lib\site-packages (from pandas) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in c:\users\sergi\anaconda3\lib\site-p
ackages (from pandas) (2022.1)
Requirement already satisfied: numpy>=1.18.5 in c:\users\sergi\anaconda3\lib\site-
packages (from pandas) (1.21.5)
Requirement already satisfied: six>=1.5 in c:\users\sergi\anaconda3\lib\site-packa
ges (from python-dateutil>=2.8.1->pandas) (1.16.0)
```

In [2]:
```
# And to import pandas
import pandas as pd
```

Series

- A series is a one.dimensional labeled array capable of holding any data type.
- The axis labels are referred to as the index.

In [4]:
```
import numpy as np   # import numpy as well

s = pd.Series(np.arange(5),index = ["a", "b", "c", "d", "e"])
print(s)
print(type(s))   # check that it is actually a Series
```

```
a    0
b    1
c    2
d    3
e    4
dtype: int32
<class 'pandas.core.series.Series'>
```

Series from a Dictionary

- We can also create a \`Series\` from a \`dictionary\`.
- The \`keys\` will be passed as indexs
- The \`values\` as the entries of the \`Series\`.

In [5]:
```
d = {"b": 1, "a": 0, "c": 2}
s = pd.Series(d)
```

```
print(s)
```

```
b    1
a    0
c    2
dtype: int64
```

Series from scalar

In [9]:
```
s = pd.Series(6)
print(s)

# Now with a longer range
s = pd.Series(6,index=np.arange(10))
print(s)
```

```
0    6
dtype: int64
0    6
1    6
2    6
3    6
4    6
5    6
6    6
7    6
8    6
9    6
dtype: int64
```

Series properties

`Series` act very similary to a `ndarray` and is a valid argument to most NumPy functions.

In [10]:
```
print(s.index) # returns the index of the Series
print(s.values) # retunrs the values
```

```
Int64Index([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype='int64')
[6 6 6 6 6 6 6 6 6 6]
```

We can also obtain the total amount of elements from a `Series` using `Series.size` method.

In [11]:
```
print(s.size)
```

```
10
```

Indexing Series

There are two ways of accessing `Series` elements:

- Using its positional argument
- Using the index like with Python dictionaries

In [12]:
```
s = pd.Series(np.arange(3),index = ['a','b','c'])
print(s)
print(s[0])    # access using positional argument
print(s['a'])  # access using index
```

```
a    0
b    1
c    2
dtype: int32
0
0
```

Slicing works similar than in NumPy but it will also slice the index.

In [40]:
```
s  = pd.Series(np.random.randint(5,10,100))
print(s)
# slice the series
s = s[s>7]
print(s)   # Notice the index now has been also sliced!
```

```
0     9
1     9
2     9
3     9
4     7
     ..
95    6
96    5
97    9
98    9
99    7
Length: 100, dtype: int32
0     9
1     9
2     9
3     9
5     8
13    8
14    8
16    9
17    8
19    9
21    9
22    9
26    9
27    9
31    8
36    9
37    9
42    8
43    8
44    9
45    8
51    8
55    9
60    9
61    9
63    9
64    9
65    9
67    8
68    8
70    8
73    8
74    9
75    8
77    8
81    9
82    9
83    8
86    8
87    9
91    9
93    8
97    9
98    9
dtype: int32
```

Since `Series` and `ndarrays` share similar properties, it is sometimes desirable to transform `Series` to `ndarrays`. For this we use `Series.to_numpy()` method.

In [19]:
```
s = s.to_numpy()
print(type(s))
```

```
<class 'numpy.ndarray'>
```

Operations with Series

As with `ndarrays` we can perform some operations.

Sum

In [32]:
```
s1 = pd.Series(np.arange(5,10))
s2 = pd.Series(np.arange(35,40))

# Sum of two series

print(s1+s2)
print(s1.add(s2))    # we can use the .add() method
```

```
0    40
1    42
2    44
3    46
4    48
dtype: int32
0    40
1    42
2    44
3    46
4    48
dtype: int32
```

The `Series.add()` method is very useful in case we have som `NaN` values, since we can specify their new outcome.

In [34]:
```
s3 = pd.Series([1,3,4,np.nan,9])
print(s3)

print(s1+s3)   # the outcome is a nan
print(s1.add(s3,fill_value=0))  # we can change it for 0
```

```
0    1.0
1    3.0
2    4.0
3    NaN
4    9.0
dtype: float64
0     6.0
1     9.0
2    11.0
3     NaN
4    18.0
dtype: float64
0     6.0
1     9.0
2    11.0
3     8.0
4    18.0
dtype: float64
```

A key difference between `Series` and `ndarray` is that operations between `Series` automatically align the data based on label. Thus, you can write computations without giving consideration to whether the `Series` involved have the same labels.

In [38]:
```
print(s[1:]+s[:-1])  # Only sums the values that share the same index
```

```
0     NaN
1    12.0
2    14.0
3    16.0
4     NaN
dtype: float64
```

Substraction

The idea is the same as before but now using the operator `-` or the method `Series.sub()`.

In [35]:
```
# Substraction
print(s1-s2)
```

```
0   -30
1   -30
2   -30
3   -30
4   -30
dtype: int32
```

Multiplication

This is done with the operator `*` or with the method `Series.mul()`.

In [37]:
```
# Multiplication
print(s1.mul(s3,fill_value=99))
```

```
0      5.0
1     18.0
2     28.0
3    792.0
4     81.0
dtype: float64
```

Division

It is done with the operator `/` or with the method `Series.div()`.

In [39]:
```
print(s1/s2)
```

```
0    0.142857
1    0.166667
2    0.189189
3    0.210526
4    0.230769
dtype: float64
```

NaN Detection

Finally, it is possible to identify NaN values with different tools.

isnull()

Will detect the missing values of a Series and returns a Serie of booleans with True or False.

In [41]:
```
print(s3.isnull())
```

```
0    False
1    False
2    False
3     True
4    False
dtype: bool
```

notnull()

Detects the not missing values. It is the oposite to `isnull()`.

In [42]:
```
print(s3.notnull())
```

```
0     True
1     True
2     True
3    False
4     True
dtype: bool
```

DataFrame

A `DataFrame` is the main object from the Pandas library. It is a 2-dimensional labeled data structure with coluns of potentially different types.

From a Dictionary

Using this method the `keys` become the columns of the `DataFrame`.

In [44]:
```
dic = {'a':[1,2],'b':[34,456],'c':[56,65],'d':[23,23]}

df = pd.DataFrame(dic)
print(df)
```

```
   a    b   c   d
0  1   34  56  23
1  2  456  65  23
```

From a Dictionary of Series

Using this method, the `keys` of the dictionary become the columns of the `DataFrame` and the `index` of the `Series` become the `index` of the `DataFrame`.

In [46]:
```
d = {
    "one": pd.Series([1.0, 2.0, 3.0], index=["a", "b", "c"]),
    "two": pd.Series([1.0, 2.0, 3.0, 4.0], index=["a", "b", "c", "d"]),
}
```

```
df = pd.DataFrame(d)
print(df)
```

```
   one  two
a  1.0  1.0
b  2.0  2.0
c  3.0  3.0
d  NaN  4.0
```

We can also specify the index that we want. This will take a subset of the indices on the `Series`.

In [47]:
```
print(pd.DataFrame(d, index=["d", "b", "a"]))
```

```
   one  two
d  NaN  4.0
b  2.0  2.0
a  1.0  1.0
```

Finally, we can also specify the columns that we want, even if they do not exist (generating a column of missings).

In [48]:
```
pd.DataFrame(d, index=["d", "b", "a"], columns=["two", "three"])
```

Out[48]:

	two	three
d	4.0	NaN
b	2.0	NaN
a	1.0	NaN

From a NumPy array

In [50]:
```
array = np.arange(20).reshape((10,2))
pd.DataFrame(array)
```

Out[50]:

	0	1
0	0	1
1	2	3
2	4	5
3	6	7
4	8	9
5	10	11
6	12	13
7	14	15
8	16	17
9	18	19

From a Dictoinary of NumPy arrays

In [51]:
```
d = {"one": [1.0, 2.0, 3.0, 4.0], "two": [4.0, 3.0, 2.0, 1.0]}

pd.DataFrame(d)
```

Out[51]:

	one	two
0	1.0	4.0
1	2.0	3.0
2	3.0	2.0
3	4.0	1.0

We can also introduce index labels if we wish.

In [52]:
```
pd.DataFrame(d, index=["a", "b", "c", "d"])
```

Out[52]:

	one	two
a	1.0	4.0
b	2.0	3.0
c	3.0	2.0
d	4.0	1.0

Working With DataFrames

Loading Data

We will use a `read_xxx()` function.

In [3]:
```
df = pd.read_csv('Pokemon.csv')   # Load the data as a df.
```

Saving Data

We will use `to_xxx()` function.

In [63]:
```
df.to_stata('Pokemon.dta')
```

```
C:\Users\Sergi\anaconda3\lib\site-packages\pandas\io\stata.py:2491: InvalidColumnN
ame:
Not all pandas column names were valid Stata variable names.
The following replacements have been made:

    #   ->   _
    Type 1   ->   Type_1
    Type 2   ->   Type_2
    Sp. Atk   ->   Sp__Atk
    Sp. Def   ->   Sp__Def

If this is not what you expect, please make sure you have Stata-compliant
column names in your DataFrame (strings only, max 32 characters, only
alphanumerics and underscores, no Stata reserved words)

  warnings.warn(ws, InvalidColumnName)
```

Do not save the index! (Unless you want)

In [59]:
```
df.to_stata('Pokemon.dta',write_index=False)
```

```
C:\Users\Sergi\anaconda3\lib\site-packages\pandas\io\stata.py:2491: InvalidColumnN
ame:
Not all pandas column names were valid Stata variable names.
The following replacements have been made:

    #   ->   _
    Type 1   ->   Type_1
    Type 2   ->   Type_2
    Sp. Atk   ->   Sp__Atk
    Sp. Def   ->   Sp__Def

If this is not what you expect, please make sure you have Stata-compliant
column names in your DataFrame (strings only, max 32 characters, only
alphanumerics and underscores, no Stata reserved words)

  warnings.warn(ws, InvalidColumnName)
```

Exploration Methods

In [64]:
```
print(df)   # prints the data frame
```

```
       #                   Name   Type 1  Type 2  Total  HP  Attack  Defense  \
0      1              Bulbasaur    Grass  Poison    318  45      49       49   
1      2                Ivysaur    Grass  Poison    405  60      62       63   
2      3               Venusaur    Grass  Poison    525  80      82       83   
3      3  VenusaurMega Venusaur    Grass  Poison    625  80     100      123   
4      4             Charmander     Fire     NaN    309  39      52       43   
..   ...                    ...      ...     ...    ...  ..     ...      ...   
795  719                Diancie     Rock   Fairy    600  50     100      150   
796  719    DiancieMega Diancie     Rock   Fairy    700  50     160      110   
797  720    HoopaHoopa Confined  Psychic   Ghost    600  80     110       60   
798  720     HoopaHoopa Unbound  Psychic    Dark    680  80     160       60   
799  721              Volcanion     Fire   Water    600  80     110      120   

     Sp. Atk  Sp. Def  Speed  Generation  Legendary  
0         65       65     45           1      False  
1         80       80     60           1      False  
2        100      100     80           1      False  
3        122      120     80           1      False  
4         60       50     65           1      False  
..       ...      ...    ...         ...        ...  
795      100      150     50           6       True  
796      160      110    110           6       True  
797      150      130     70           6       True  
798      170      130     80           6       True  
799      130       90     70           6       True  

[800 rows x 13 columns]
```

df.head()

Print the first `X` rows.

In [4]:
```
df.head(5)  # display the first 5 rows
```

Out[4]:

	#	Name	Type 1	Type 2	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Le
0	1	Bulbasaur	Grass	Poison	318	45	49	49	65	65	45	1	
1	2	Ivysaur	Grass	Poison	405	60	62	63	80	80	60	1	
2	3	Venusaur	Grass	Poison	525	80	82	83	100	100	80	1	
3	3	VenusaurMega Venusaur	Grass	Poison	625	80	100	123	122	120	80	1	
4	4	Charmander	Fire	NaN	309	39	52	43	60	50	65	1	

df.tail()

Equivalently, we can display the last amount of rows that we want. Notice that the column names will still appear.

In [66]:
```
df.tail(3)  # display the last 3 rows
```

Out[66]:

	#	Name	Type 1	Type 2	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation
797	720	HoopaHoopa Confined	Psychic	Ghost	600	80	110	60	150	130	70	6
798	720	HoopaHoopa Unbound	Psychic	Dark	680	80	160	60	170	130	80	6
799	721	Volcanion	Fire	Water	600	80	110	120	130	90	70	6

df.sample()

If we don't trust the first or the last rows, we can still display a random sample of our data with the amount of rows that we want.

In [67]:
```
df.sample(6)
```

Out[67]:

	#	Name	Type 1	Type 2	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation
302	279	Pelipper	Water	Flying	430	60	50	100	85	70	65	3
433	388	Grotle	Grass	NaN	405	75	89	85	55	65	36	4
439	394	Prinplup	Water	NaN	405	64	66	68	81	76	50	4
210	195	Quagsire	Water	Ground	430	95	85	85	65	65	35	2
2	3	Venusaur	Grass	Poison	525	80	82	83	100	100	80	1
282	260	Swampert	Water	Ground	535	100	110	90	85	90	60	3

By taking a look at the index, we can see that the rows are totally random, and also they are unordered.

df.shape

If we are willing to learn the number of rows and columns our `DataFrame` has.

In [69]:
```
df.shape   #(rows,columns)
```

Out[69]:
```
(800, 13)
```

df.dtypes

It will tell us the type of each of the variables of our `DataFrame`.

In [71]:
```
df.dtypes
```

Out[71]:
```
#               int64
Name           object
Type 1         object
Type 2         object
Total           int64
HP              int64
Attack          int64
Defense         int64
Sp. Atk         int64
Sp. Def         int64
Speed           int64
Generation      int64
Legendary        bool
dtype: object
```

df.columns

Suppose we are only interested in the column names of our `DataFrame`.

In [73]:
```
df.columns
```

Out[73]:
```
Index(['#', 'Name', 'Type 1', 'Type 2', 'Total', 'HP', 'Attack', 'Defense',
       'Sp. Atk', 'Sp. Def', 'Speed', 'Generation', 'Legendary'],
      dtype='object')
```

df.describe()

This will return a `DataFrame` with the summary statistics from our numeric columns.

In [74]:
```
df.describe()
```

Out[74]:

	#	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Sp
count	800.000000	800.00000	800.000000	800.000000	800.000000	800.000000	800.000000	800.000
mean	362.813750	435.10250	69.258750	79.001250	73.842500	72.820000	71.902500	68.277
std	208.343798	119.96304	25.534669	32.457366	31.183501	32.722294	27.828916	29.060
min	1.000000	180.00000	1.000000	5.000000	5.000000	10.000000	20.000000	5.000
25%	184.750000	330.00000	50.000000	55.000000	50.000000	49.750000	50.000000	45.000
50%	364.500000	450.00000	65.000000	75.000000	70.000000	65.000000	70.000000	65.000
75%	539.250000	515.00000	80.000000	100.000000	90.000000	95.000000	90.000000	90.000
max	721.000000	780.00000	255.000000	190.000000	230.000000	194.000000	230.000000	180.000

df.info()

Will return a little summary of our `DataFrame`.

In [75]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 800 entries, 0 to 799
Data columns (total 13 columns):
 #   Column      Non-Null Count  Dtype 
---  ------      --------------  ----- 
 0   #           800 non-null    int64 
 1   Name        800 non-null    object
 2   Type 1      800 non-null    object
 3   Type 2      414 non-null    object
 4   Total       800 non-null    int64 
 5   HP          800 non-null    int64 
 6   Attack      800 non-null    int64 
 7   Defense     800 non-null    int64 
 8   Sp. Atk     800 non-null    int64 
 9   Sp. Def     800 non-null    int64 
 10  Speed       800 non-null    int64 
 11  Generation  800 non-null    int64 
 12  Legendary   800 non-null    bool  
dtypes: bool(1), int64(9), object(3)
memory usage: 75.9+ KB
```

df.unique()

Returns unique elements.

In [76]:
```
df['Generation'].unique()
```

Out[76]:
```
array([1, 2, 3, 4, 5, 6], dtype=int64)
```

Notice that to refer to a particular `DataFrame` column we are using the column name. We can also subset based on rows.

In [78]:
```
df['Generation'][:6] # First 6 rows from column generation
```

Out[78]:
```
0    1
1    1
2    1
3    1
4    1
5    1
Name: Generation, dtype: int64
```

df.value_counts()

It returns a `Serie` with the unique values and the amount of times that they appear.

In [79]:
```
df['Type 1'].value_counts()
```

Out[79]:
```
Water       112
Normal       98
Grass        70
Bug          69
Psychic      57
Fire         52
Electric     44
Rock         44
Dragon       32
Ground       32
Ghost        32
Dark         31
Poison       28
Steel        27
Fighting     27
Ice          24
Fairy        17
Flying        4
Name: Type 1, dtype: int64
```

df.isnull()

Will return `True` if missing and `False` if not.

In [ ]:
```
df.isnull()
```

To make it easier to understand, we can sum the result from `isnull()` to see the amount of missing observations at every column.

In [82]:
```
df.isnull().sum()
```

Out[82]:
```
#               0
Name            0
Type 1          0
Type 2        386
Total           0
HP              0
Attack          0
Defense         0
Sp. Atk         0
Sp. Def         0
Speed           0
Generation      0
Legendary       0
dtype: int64
```

Indexing and Slicing DataFrames

Pandas offer a lot of opportunities to index and slice their main object, `DataFrames`. The main object to do that are `loc` and `iloc`. The main differences are that:

- **loc** is label-based, which means that you have to specify rows and columns based on their row and columb labels.
- **iloc** is integer position-based, so you have to specify rows and columns based by their integer position values.

So illustrate how they work let us estalish a different index in our `DataFrame`. To do that we will use the module `set_index()`.

In [101…

```python
df.set_index('Name',inplace=True)
df.head(3)
```

Out[101]:

	#	Type 1	Type 2	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Legendary
Name												
Bulbasaur	1	Grass	Poison	318	45	49	49	65	65	45	1	False
Ivysaur	2	Grass	Poison	405	60	62	63	80	80	60	1	False
Venusaur	3	Grass	Poison	525	80	82	83	100	100	80	1	False

Now the index corresponds with the name of the Pokemon.

Locate a column

The easiest way to refer to a column is just using its name. We do not need to use `iloc` or `loc`, we can just type `df['colname']`.

In [102…

```python
print(df['Type 1'])
```

```
Name
Bulbasaur                    Grass
Ivysaur                      Grass
Venusaur                     Grass
VenusaurMega Venusaur        Grass
Charmander                    Fire
                            ...
Diancie                       Rock
DiancieMega Diancie           Rock
HoopaHoopa Confined        Psychic
HoopaHoopa Unbound         Psychic
Volcanion                     Fire
Name: Type 1, Length: 800, dtype: object
```

In [104…

```python
print(df.loc[:,'Type 1'])  # Locate using loc
```

```
Name
Bulbasaur                    Grass
Ivysaur                      Grass
Venusaur                     Grass
VenusaurMega Venusaur        Grass
Charmander                    Fire
                            ...
Diancie                       Rock
DiancieMega Diancie           Rock
HoopaHoopa Confined        Psychic
HoopaHoopa Unbound         Psychic
Volcanion                     Fire
Name: Type 1, Length: 800, dtype: object
```

In [103…

```python
print(df.iloc[:,2])  # Locate using iloc
```

```
Name
Bulbasaur                   Poison
Ivysaur                     Poison
Venusaur                    Poison
VenusaurMega Venusaur       Poison
Charmander                     NaN
                             ...  
Diancie                      Fairy
DiancieMega Diancie          Fairy
HoopaHoopa Confined          Ghost
HoopaHoopa Unbound            Dark
Volcanion                    Water
Name: Type 2, Length: 800, dtype: object
```

Locate multiple Columns

To do that we can do:

In [111…

```
print(df[['Type 1','Type 2']])
```

```
                        Type 1  Type 2
Name                                  
Bulbasaur                Grass  Poison
Ivysaur                  Grass  Poison
Venusaur                 Grass  Poison
VenusaurMega Venusaur    Grass  Poison
Charmander                Fire     NaN
...                        ...     ...
Diancie                   Rock   Fairy
DiancieMega Diancie       Rock   Fairy
HoopaHoopa Confined    Psychic   Ghost
HoopaHoopa Unbound     Psychic    Dark
Volcanion                 Fire   Water

[800 rows x 2 columns]
```

Or we can use the `iloc` and the `loc` commands.

In [114…

```
df.loc[:,['Type 1','Type 2']]
```

Out[114]:

Name	Type 1	Type 2
Bulbasaur	Grass	Poison
Ivysaur	Grass	Poison
Venusaur	Grass	Poison
VenusaurMega Venusaur	Grass	Poison
Charmander	Fire	NaN
...	...	...
Diancie	Rock	Fairy
DiancieMega Diancie	Rock	Fairy
HoopaHoopa Confined	Psychic	Ghost
HoopaHoopa Unbound	Psychic	Dark
Volcanion	Fire	Water

800 rows × 2 columns

In [116…

```
df.iloc[:,[1,2]]
```

Out[116]:

Name	Type 1	Type 2
Bulbasaur	Grass	Poison
Ivysaur	Grass	Poison
Venusaur	Grass	Poison
VenusaurMega Venusaur	Grass	Poison
Charmander	Fire	NaN
...	...	...
Diancie	Rock	Fairy
DiancieMega Diancie	Rock	Fairy
HoopaHoopa Confined	Psychic	Ghost
HoopaHoopa Unbound	Psychic	Dark
Volcanion	Fire	Water

800 rows × 2 columns

Locate a row

To locate a row or a slice of rows we can only use `iloc` or `loc`.

In [105…

```
df.iloc[2:6]    # Locate the rows 2 to 5.
```

Out[105]:

Name	#	Type 1	Type 2	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Lege
Venusaur	3	Grass	Poison	525	80	82	83	100	100	80	1	
VenusaurMega Venusaur	3	Grass	Poison	625	80	100	123	122	120	80	1	
Charmander	4	Fire	NaN	309	39	52	43	60	50	65	1	
Charmeleon	5	Fire	NaN	405	58	64	58	80	65	80	1	

In [109…

```
df.loc['Venusaur':'Charmeleon'] # locate the rows 2 to 5
```

Out[109]:

Name	#	Type 1	Type 2	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Lege
Venusaur	3	Grass	Poison	525	80	82	83	100	100	80	1	
VenusaurMega Venusaur	3	Grass	Poison	625	80	100	123	122	120	80	1	
Charmander	4	Fire	NaN	309	39	52	43	60	50	65	1	
Charmeleon	5	Fire	NaN	405	58	64	58	80	65	80	1	

Notice here we can spot an important difference among the two methods. If we want to select rows two to five, when using `iloc` we specify the interval `[2:6]` since row `6` is the first element that is **not** included. However, when locating the same interval using `loc` we specify `['Venusaur':'Charmeleon']` since `Charmeleon` is the last **included** element.

Locate Rows and Columns

In [117…

```
df.iloc[1,[2,3]]    # row 1, columns 2 and 3
```

Out[117]:

```
Type 2    Poison
Total        405
Name: Ivysaur, dtype: object
```

In [119…

```
df.loc['Ivysaur',['Type 2','Total']]  # row 1, columns 2 and 3
```

Out[119]:

```
Type 2    Poison
Total        405
Name: Ivysaur, dtype: object
```

Finally, we can also locate an specific element.

In [120…

```
df.loc['Ivysaur','Type 2']
```

Out[120]:

```
'Poison'
```

In [121…

```
df.iloc[1,2]
```

Out[121]: 'Poison'

Slicing based on logical conditions

We can also apply logical conditions to get slices from the `DataFrame` object.

One condition

In [129…

```
# using the loc method
df.loc[df['Type 2'] == 'Water']
```

Out[129]:

	#	Type 1	Type 2	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Leg
Name												
Omanyte	138	Rock	Water	355	35	40	100	90	55	35	1	
Omastar	139	Rock	Water	495	70	60	125	115	70	55	1	
Kabuto	140	Rock	Water	355	30	80	90	55	45	55	1	
Kabutops	141	Rock	Water	495	60	115	105	65	70	80	1	
Surskit	283	Bug	Water	269	40	30	32	50	52	65	3	
Spheal	363	Ice	Water	290	70	40	50	55	50	25	3	
Sealeo	364	Ice	Water	410	90	60	70	75	70	45	3	
Walrein	365	Ice	Water	530	110	80	90	95	90	65	3	
Bibarel	400	Normal	Water	410	79	85	60	55	60	71	4	
RotomWash Rotom	479	Electric	Water	520	50	65	107	105	107	86	4	
Binacle	688	Rock	Water	306	42	52	67	39	56	50	6	
Barbaracle	689	Rock	Water	500	72	105	115	54	86	68	6	
Skrelp	690	Poison	Water	320	50	60	60	60	60	30	6	
Volcanion	721	Fire	Water	600	80	110	120	130	90	70	6	

In [128…

```
df[df['Type 2'] == 'Water']
```

Out[128]:

Name	#	Type 1	Type 2	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Leg
Omanyte	138	Rock	Water	355	35	40	100	90	55	35	1	
Omastar	139	Rock	Water	495	70	60	125	115	70	55	1	
Kabuto	140	Rock	Water	355	30	80	90	55	45	55	1	
Kabutops	141	Rock	Water	495	60	115	105	65	70	80	1	
Surskit	283	Bug	Water	269	40	30	32	50	52	65	3	
Spheal	363	Ice	Water	290	70	40	50	55	50	25	3	
Sealeo	364	Ice	Water	410	90	60	70	75	70	45	3	
Walrein	365	Ice	Water	530	110	80	90	95	90	65	3	
Bibarel	400	Normal	Water	410	79	85	60	55	60	71	4	
RotomWash Rotom	479	Electric	Water	520	50	65	107	105	107	86	4	
Binacle	688	Rock	Water	306	42	52	67	39	56	50	6	
Barbaracle	689	Rock	Water	500	72	105	115	54	86	68	6	
Skrelp	690	Poison	Water	320	50	60	60	60	60	30	6	
Volcanion	721	Fire	Water	600	80	110	120	130	90	70	6	

Multiple Conditions

We can also filter based on multiple conditions. Those can be for the same column or for different ones.

In [131…

```
df.loc[(df['Type 1'] == 'Water') | (df['Type 1'] == 'Fire')]
```

Out[131]:

Name	#	Type 1	Type 2	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	L
Charmander	4	Fire	NaN	309	39	52	43	60	50	65	1	
Charmeleon	5	Fire	NaN	405	58	64	58	80	65	80	1	
Charizard	6	Fire	Flying	534	78	84	78	109	85	100	1	
CharizardMega Charizard X	6	Fire	Dragon	634	78	130	111	130	85	100	1	
CharizardMega Charizard Y	6	Fire	Flying	634	78	104	78	159	115	100	1	
...	...	...	...	...	...	...	...	...	...	...	...	
Litleo	667	Fire	Normal	369	62	50	58	73	54	72	6	
Pyroar	668	Fire	Normal	507	86	68	72	109	66	106	6	
Clauncher	692	Water	NaN	330	50	53	62	58	63	44	6	
Clawitzer	693	Water	NaN	500	71	73	88	120	89	59	6	
Volcanion	721	Fire	Water	600	80	110	120	130	90	70	6	

164 rows × 12 columns

Importantly, when filtering in `DataFrames` based on multiple conditions we will use the `&` rather than the `and` or the `|` rather than the `or`.

In [6]:

```
# We can condition on two different columns

df.loc[(df['Type 1'] == 'Fire') & (df['Attack']>100)]
```

Out[6]:

	#	Name	Type 1	Type 2	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Gen
7	6	CharizardMega Charizard X	Fire	Dragon	634	78	130	111	130	85	100	
8	6	CharizardMega Charizard Y	Fire	Flying	634	78	104	78	159	115	100	
64	59	Arcanine	Fire	NaN	555	90	110	80	100	80	95	
147	136	Flareon	Fire	NaN	525	65	130	60	95	110	65	
263	244	Entei	Fire	NaN	580	115	115	85	90	75	100	
270	250	Ho-oh	Fire	Flying	680	106	130	90	110	154	90	
278	257	Blaziken	Fire	Fighting	530	80	120	70	110	70	80	
279	257	BlazikenMega Blaziken	Fire	Fighting	630	80	160	80	130	80	100	
354	323	CameruptMega Camerupt	Fire	Ground	560	70	120	100	145	105	20	
437	392	Infernape	Fire	Fighting	534	76	104	71	104	71	108	
559	500	Emboar	Fire	Fighting	528	110	123	65	100	65	65	
615	555	DarmanitanStandard Mode	Fire	NaN	480	105	140	55	30	55	95	
799	721	Volcanion	Fire	Water	600	80	110	120	130	90	70	

Notice we can include as much conditions as we want.

In [136…

```
df.loc[(df['Type 1'] == 'Water') & (df['Attack']>100) & (df['Speed'] >80)]
```

Out[136]:

Name	#	Type 1	Type 2	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	l
Gyarados	130	Water	Flying	540	95	125	79	60	100	81	1	
GyaradosMega Gyarados	130	Water	Dark	640	95	155	109	70	130	81	1	
Sharpedo	319	Water	Dark	460	70	120	40	95	40	95	3	
SharpedoMega Sharpedo	319	Water	Dark	560	70	140	70	110	65	105	3	
KyogrePrimal Kyogre	382	Water	NaN	770	100	150	90	180	160	90	3	
Floatzel	419	Water	NaN	495	85	105	55	85	50	115	4	
Palkia	484	Water	Dragon	680	90	120	100	150	120	100	4	

Filtering using pandas specific methods

df.isin()

It allows to filter based on wether a column contains values that we specify.

In [177…

```
df[df['Type 1'].isin(['Ice','Grass'])]
```

Out[177]:

Name	index	#	Type 1	Type 2	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation
Bulbasaur	0	1	Grass	Poison	1.590	45	0	49	65	65	45	1
Oddish	48	43	Grass	Poison	1.600	45	0	55	75	65	30	1
Gloom	49	44	Grass	Poison	1.975	60	0	70	85	75	40	1
Bellsprout	75	69	Grass	Poison	1.500	50	0	35	70	30	40	1
Weepinbell	76	70	Grass	Poison	1.950	65	0	50	85	45	55	1
Exeggcute	110	102	Grass	Psychic	1.625	60	0	80	60	45	40	1
Chikorita	166	152	Grass	NaN	1.590	45	0	65	49	65	45	2
Hoppip	202	187	Grass	Flying	1.250	35	0	40	35	55	50	2
Skiploom	203	188	Grass	Flying	1.700	55	0	50	45	65	80	2
Sunkern	206	191	Grass	NaN	0.900	30	0	30	30	30	30	2
Swinub	238	220	Ice	Ground	1.250	50	0	40	30	30	50	2
Delibird	243	225	Ice	Flying	1.650	45	0	45	65	45	75	2
Smoochum	257	238	Ice	Psychic	1.525	45	0	15	85	65	65	2
Treecko	272	252	Grass	NaN	1.550	40	0	35	65	55	70	3
Seedot	296	273	Grass	NaN	1.100	40	0	50	30	30	30	3
Nuzleaf	297	274	Grass	Dark	1.700	70	0	40	60	40	60	3
Shroomish	309	285	Grass	NaN	1.475	60	0	60	40	60	35	3
Roselia	344	315	Grass	Poison	2.000	50	0	45	100	80	65	3
Cacnea	362	331	Grass	NaN	1.675	50	0	40	85	40	35	3
Snorunt	395	361	Ice	NaN	1.500	50	0	50	50	50	50	3
Spheal	398	363	Ice	Water	1.450	70	0	50	55	50	25	3
Turtwig	432	387	Grass	NaN	1.590	55	0	64	45	55	31	4
Budew	451	406	Grass	Poison	1.400	40	0	35	50	70	55	4
Cherubi	467	420	Grass	NaN	1.375	45	0	45	62	53	35	4
Snover	509	459	Grass	Ice	1.670	60	0	50	62	60	40	4
Snivy	554	495	Grass	NaN	1.540	45	0	55	45	55	63	5
Pansage	570	511	Grass	NaN	1.580	50	0	48	53	48	64	5
Cottonee	606	546	Grass	Fairy	1.400	40	0	60	37	50	66	5
Petilil	608	548	Grass	NaN	1.400	45	0	50	70	50	30	5
Vanillite	643	582	Ice	NaN	1.525	36	0	50	65	60	44	5
Vanillish	644	583	Ice	NaN	1.975	51	0	65	80	75	59	5
Foongus	651	590	Grass	Poison	1.470	69	0	45	55	55	15	5
Ferroseed	658	597	Grass	Steel	1.525	44	0	91	24	86	10	5
Cubchoo	674	613	Ice	NaN	1.525	55	0	40	60	40	40	5

Name	index	#	Type 1	Type 2	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation
Chespin	718	650	Grass	NaN	1.565	56	0	65	48	45	38	6
Skiddo	740	672	Grass	NaN	1.750	66	0	48	62	57	52	6
Bergmite	788	712	Ice	NaN	1.520	55	0	85	32	35	28	6

Iterating through Rows

We can also iterate though each row using the `interrows()` method.

In [139...

```
for index,row in df[:5].iterrows():   # only first 4 rows
    print(index,row)
```

```
Bulbasaur #                  1
Type 1         Grass
Type 2        Poison
Total            318
HP                45
Attack            49
Defense           49
Sp. Atk           65
Sp. Def           65
Speed             45
Generation         1
Legendary      False
Name: Bulbasaur, dtype: object
Ivysaur #                  2
Type 1         Grass
Type 2        Poison
Total            405
HP                60
Attack            62
Defense           63
Sp. Atk           80
Sp. Def           80
Speed             60
Generation         1
Legendary      False
Name: Ivysaur, dtype: object
Venusaur #                  3
Type 1         Grass
Type 2        Poison
Total            525
HP                80
Attack            82
Defense           83
Sp. Atk          100
Sp. Def          100
Speed             80
Generation         1
Legendary      False
Name: Venusaur, dtype: object
VenusaurMega Venusaur #                  3
Type 1         Grass
Type 2        Poison
Total            625
HP                80
Attack           100
Defense          123
Sp. Atk          122
Sp. Def          120
Speed             80
Generation         1
Legendary      False
Name: VenusaurMega Venusaur, dtype: object
Charmander #                  4
Type 1         Fire
Type 2          NaN
Total           309
HP               39
Attack           52
Defense          43
Sp. Atk          60
Sp. Def          50
Speed            65
Generation        1
```

```
Legendary      False
Name: Charmander, dtype: object
```

Modifying Characteristics

Methods to modify features

df.set_index()

Allows to change the index of the `DataFrame`.

In [141…
```python
df.set_index('Type 2') # set type 2 as the index.
```

Out[141]:

	#	Name	Type 1	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	L
Type 2												
Poison	1	Bulbasaur	Grass	318	45	49	49	65	65	45	1	
Poison	2	Ivysaur	Grass	405	60	62	63	80	80	60	1	
Poison	3	Venusaur	Grass	525	80	82	83	100	100	80	1	
Poison	3	VenusaurMega Venusaur	Grass	625	80	100	123	122	120	80	1	
NaN	4	Charmander	Fire	309	39	52	43	60	50	65	1	
...	...	...	...	...	...	...	...	...	...	...	...	
Fairy	719	Diancie	Rock	600	50	100	150	100	150	50	6	
Fairy	719	DiancieMega Diancie	Rock	700	50	160	110	160	110	110	6	
Ghost	720	HoopaHoopa Confined	Psychic	600	80	110	60	150	130	70	6	
Dark	720	HoopaHoopa Unbound	Psychic	680	80	160	60	170	130	80	6	
Water	721	Volcanion	Fire	600	80	110	120	130	90	70	6	

800 rows × 12 columns

In [142…
```python
df.set_index(['Name','Type 1'])   # Set multiple variables as our index.
```

Out[142]:

Name	Type 1	#	Type 2	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation
Bulbasaur	**Grass**	1	Poison	318	45	49	49	65	65	45	1
Ivysaur	**Grass**	2	Poison	405	60	62	63	80	80	60	1
Venusaur	**Grass**	3	Poison	525	80	82	83	100	100	80	1
VenusaurMega Venusaur	**Grass**	3	Poison	625	80	100	123	122	120	80	1
Charmander	**Fire**	4	NaN	309	39	52	43	60	50	65	1
...	**...**	...	...	...	...	...	...	...	...	...	...
Diancie	**Rock**	719	Fairy	600	50	100	150	100	150	50	6
DiancieMega Diancie	**Rock**	719	Fairy	700	50	160	110	160	110	110	6
HoopaHoopa Confined	**Psychic**	720	Ghost	600	80	110	60	150	130	70	6
HoopaHoopa Unbound	**Psychic**	720	Dark	680	80	160	60	170	130	80	6
Volcanion	**Fire**	721	Water	600	80	110	120	130	90	70	6

800 rows × 11 columns

Notice however, that we are creating an object, but not changing the original `DataFrame`. If we want to change the orinigal `DataFrame` we need to use the `inplace` argument.

In [149...

```
print(df.head(5))
df.set_index(['Name'],inplace=True)
print(df.head(5))
```

```
   index                   Name  # Type 1  Type 2  Total  HP  Attack  Defense  \
0      0              Bulbasaur  1  Grass  Poison    318  45      49       49
1      1                Ivysaur  2  Grass  Poison    405  60      62       63
2      2               Venusaur  3  Grass  Poison    525  80      82       83
3      3  VenusaurMega Venusaur  3  Grass  Poison    625  80     100      123
4      4             Charmander  4   Fire     NaN    309  39      52       43

   Sp. Atk  Sp. Def  Speed  Generation  Legendary
0       65       65     45           1      False
1       80       80     60           1      False
2      100      100     80           1      False
3      122      120     80           1      False
4       60       50     65           1      False
                       index  # Type 1  Type 2  Total  HP  Attack  Defense  \
Name
Bulbasaur                  0  1  Grass  Poison    318  45      49       49
Ivysaur                    1  2  Grass  Poison    405  60      62       63
Venusaur                   2  3  Grass  Poison    525  80      82       83
VenusaurMega Venusaur      3  3  Grass  Poison    625  80     100      123
Charmander                 4  4   Fire     NaN    309  39      52       43

                       Sp. Atk  Sp. Def  Speed  Generation  Legendary
Name
Bulbasaur                   65       65     45           1      False
Ivysaur                     80       80     60           1      False
Venusaur                   100      100     80           1      False
VenusaurMega Venusaur      122      120     80           1      False
Charmander                  60       50     65           1      False
```

df.reset_index()

We can also remove the index we have established and go back to the initial configuration.

In [148…

```
df.reset_index(inplace=True)
print(df.head(3))
```

```
   index       Name  # Type 1  Type 2  Total  HP  Attack  Defense  Sp. Atk  \
0      0  Bulbasaur  1  Grass  Poison    318  45      49       49       65
1      1    Ivysaur  2  Grass  Poison    405  60      62       63       80
2      2   Venusaur  3  Grass  Poison    525  80      82       83      100

   Sp. Def  Speed  Generation  Legendary
0       65     45           1      False
1       80     60           1      False
2      100     80           1      False
```

df.rename()

It will allow us to change the column names. It takes a dictionary as an input.

In [150…

```
df.rename(columns={'Attack':'Strength'},inplace=True)
df.head(2)
```

Out[150]:

Name	index	#	Type 1	Type 2	Total	HP	Strength	Defense	Sp. Atk	Sp. Def	Speed	Generation
Bulbasaur	0	1	Grass	Poison	318	45	49	49	65	65	45	1
Ivysaur	1	2	Grass	Poison	405	60	62	63	80	80	60	1

In [151…

```
df.rename(columns={'Strength':'Attack'},inplace=True)    # Change it back
```

df.sort_values()

It will sort the `DataFrame` based on one or multiple columns according to the numerical or alphabetical order.

In [153…

```
df.sort_values('Type 1')
```

Out[153]:

Name	index	#	Type 1	Type 2	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation
Sewaddle	600	540	Bug	Grass	310	45	53	70	40	60	42	5
Pinsir	136	127	Bug	NaN	500	65	125	100	55	70	85	1
Burmy	457	412	Bug	NaN	224	40	29	45	29	45	36	4
Scyther	132	123	Bug	Flying	500	70	110	80	55	80	105	1
Joltik	656	595	Bug	Electric	319	50	47	50	57	50	65	5
...	...	...	...	...	...	...	...	...	...	...	...	...
Totodile	172	158	Water	NaN	314	50	65	64	44	48	43	2
Basculin	610	550	Water	NaN	460	70	92	65	80	55	98	5
Vaporeon	145	134	Water	NaN	525	130	65	60	110	95	65	1
Panpour	574	515	Water	NaN	316	50	53	48	53	48	64	5
Chinchou	184	170	Water	Electric	330	75	38	38	56	56	67	2

800 rows × 13 columns

We can also reverse the order by using the argument `ascending = False`.

In [154…

```
df.sort_values('Speed',ascending =False)
```

Out[154]:

Name	index	#	Type 1	Type 2	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Gene
DeoxysSpeed Forme	431	386	Psychic	NaN	600	50	95	90	95	90	180	
Ninjask	315	291	Bug	Flying	456	61	90	45	50	50	160	
DeoxysNormal Forme	428	386	Psychic	NaN	600	50	150	50	150	50	150	
AerodactylMega Aerodactyl	154	142	Rock	Flying	615	80	135	85	70	95	150	
AlakazamMega Alakazam	71	65	Psychic	NaN	590	55	50	65	175	95	150	
...	...	...	...	...	...	...	...	...	...	...	...	
Ferroseed	658	597	Grass	Steel	305	44	50	91	24	86	10	
Bonsly	486	438	Rock	NaN	290	50	80	95	10	45	10	
Trapinch	359	328	Ground	NaN	290	45	100	45	45	45	10	
Shuckle	230	213	Bug	Rock	505	20	10	230	10	230	5	
Munchlax	495	446	Normal	NaN	390	135	85	40	40	85	5	

800 rows × 13 columns

In [155…

```
#Sort based on multiple columns

df.sort_values(['Attack','Defense'], ascending = [True,False])  # First ascending,
```

Out[155]:

Name	index	#	Type 1	Type 2	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Gen
Chansey	121	113	Normal	NaN	450	250	5	5	35	105	50	
Happiny	488	440	Normal	NaN	220	100	5	5	15	65	30	
Shuckle	230	213	Bug	Rock	505	20	10	230	10	230	5	
Magikarp	139	129	Water	NaN	200	20	10	55	15	20	80	
Blissey	261	242	Normal	NaN	540	255	10	10	75	135	55	
...	...	...	...	...	...	...	...	...	...	...	...	
GroudonPrimal Groudon	424	383	Ground	Fire	770	100	180	160	150	90	90	
RayquazaMega Rayquaza	426	384	Dragon	Flying	780	105	180	100	180	100	115	
DeoxysAttack Forme	429	386	Psychic	NaN	600	50	180	20	180	20	150	
HeracrossMega Heracross	232	214	Bug	Fighting	600	80	185	115	40	105	75	
MewtwoMega Mewtwo X	163	150	Psychic	Fighting	780	106	190	100	154	100	130	

800 rows × 13 columns

df.drop()

It allows to delete a particular row or column.

In [158…

```
df.drop(['Type 2'],axis=1)    # delete variable 'Type 2'
```

Out[158]:

Name	index	#	Type 1	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Le
Bulbasaur	0	1	Grass	318	45	49	49	65	65	45	1	
Ivysaur	1	2	Grass	405	60	62	63	80	80	60	1	
Venusaur	2	3	Grass	525	80	82	83	100	100	80	1	
VenusaurMega Venusaur	3	3	Grass	625	80	100	123	122	120	80	1	
Charmander	4	4	Fire	309	39	52	43	60	50	65	1	
...	...	...	...	...	...	...	...	...	...	...	...	
Diancie	795	719	Rock	600	50	100	150	100	150	50	6	
DiancieMega Diancie	796	719	Rock	700	50	160	110	160	110	110	6	
HoopaHoopa Confined	797	720	Psychic	600	80	110	60	150	130	70	6	
HoopaHoopa Unbound	798	720	Psychic	680	80	160	60	170	130	80	6	
Volcanion	799	721	Fire	600	80	110	120	130	90	70	6	

800 rows × 12 columns

In [159…

```
df.drop(['Bulbasaur'],axis=0)  # delete the first row
```

Out[159]:

Name	index	#	Type 1	Type 2	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Genera
Ivysaur	1	2	Grass	Poison	405	60	62	63	80	80	60	
Venusaur	2	3	Grass	Poison	525	80	82	83	100	100	80	
VenusaurMega Venusaur	3	3	Grass	Poison	625	80	100	123	122	120	80	
Charmander	4	4	Fire	NaN	309	39	52	43	60	50	65	
Charmeleon	5	5	Fire	NaN	405	58	64	58	80	65	80	
...	...	...	...	...	...	...	...	...	...	...	...	
Diancie	795	719	Rock	Fairy	600	50	100	150	100	150	50	
DiancieMega Diancie	796	719	Rock	Fairy	700	50	160	110	160	110	110	
HoopaHoopa Confined	797	720	Psychic	Ghost	600	80	110	60	150	130	70	
HoopaHoopa Unbound	798	720	Psychic	Dark	680	80	160	60	170	130	80	
Volcanion	799	721	Fire	Water	600	80	110	120	130	90	70	

799 rows × 13 columns

Creating new columns and modiying existing ones

To create a new column we just need to type `df['newcol'] =` and then set the column equal to whathever we want. This can be a `Series` , a scalar or a combination of other columns of the `DataFrame` .

In [160…

```
# Creating a new column equal to scalar
df['New Column'] = 5
df.head(3)
```

Out[160]:

Name	index	#	Type 1	Type 2	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Le
Bulbasaur	0	1	Grass	Poison	318	45	49	49	65	65	45	1	
Ivysaur	1	2	Grass	Poison	405	60	62	63	80	80	60	1	
Venusaur	2	3	Grass	Poison	525	80	82	83	100	100	80	1	

In [162…

```
# Creating a new column equal to a combination of other columns

df['Total Without HP'] = df['Total'] - df['HP']
df.head(3)
```

Out[162]:

Name	index	#	Type 1	Type 2	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Le
Bulbasaur	0	1	Grass	Poison	318	45	49	49	65	65	45	1	
Ivysaur	1	2	Grass	Poison	405	60	62	63	80	80	60	1	
Venusaur	2	3	Grass	Poison	525	80	82	83	100	100	80	1	

In [164…

```
# We can also sum multiple columns to create a new column
df['Sum'] = df.iloc[:,5:8].sum(axis=1)   # sum across columns
df.head(3)
```

Out[164]:

Name	index	#	Type 1	Type 2	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Le
Bulbasaur	0	1	Grass	Poison	318	45	49	49	65	65	45	1	
Ivysaur	1	2	Grass	Poison	405	60	62	63	80	80	60	1	
Venusaur	2	3	Grass	Poison	525	80	82	83	100	100	80	1	

We can even modify an already existing column.

In [165…

```
df['Total'] = df['Total'] / 200
df.head(3)
```

Out[165]:

Name	index	#	Type 1	Type 2	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Le
Bulbasaur	0	1	Grass	Poison	1.590	45	49	49	65	65	45	1	
Ivysaur	1	2	Grass	Poison	2.025	60	62	63	80	80	60	1	
Venusaur	2	3	Grass	Poison	2.625	80	82	83	100	100	80	1	

Conditional Changes

This will allow us to change the values of some variables based on the values of another variables.

Change one variable based on itself.

In [167…

```
df.loc[df['Type 1'] == 'Water','Type 1'] = 'Wet'
df['Type 1'].unique()
```

Out[167]:
```
array(['Grass', 'Fire', 'Wet', 'Bug', 'Normal', 'Poison', 'Electric',
       'Ground', 'Fairy', 'Fighting', 'Psychic', 'Rock', 'Ghost', 'Ice',
       'Dragon', 'Dark', 'Steel', 'Flying'], dtype=object)
```

Change one variable based on another.

In [169…
```
df.loc[df['Type 1'] == 'Fire','Total'] = 999
df.head(6)
```

Out[169]:

Name	index	#	Type 1	Type 2	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generat
Bulbasaur	0	1	Grass	Poison	1.590	45	49	49	65	65	45	
Ivysaur	1	2	Grass	Poison	2.025	60	62	63	80	80	60	
Venusaur	2	3	Grass	Poison	2.625	80	82	83	100	100	80	
VenusaurMega Venusaur	3	3	Grass	Poison	3.125	80	100	123	122	120	80	
Charmander	4	4	Fire	NaN	999.000	39	52	43	60	50	65	
Charmeleon	5	5	Fire	NaN	999.000	58	64	58	80	65	80	

Change one variable based on multiple conditions

In [172…
```
df.loc[(df['Attack']>60) & (df['Attack']<1000),'Attack'] = 0
df.head(6)
```

Out[172]:

Name	index	#	Type 1	Type 2	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generat
Bulbasaur	0	1	Grass	Poison	1.590	45	0	49	65	65	45	
Ivysaur	1	2	Grass	Poison	2.025	60	0	63	80	80	60	
Venusaur	2	3	Grass	Poison	2.625	80	0	83	100	100	80	
VenusaurMega Venusaur	3	3	Grass	Poison	3.125	80	0	123	122	120	80	
Charmander	4	4	Fire	NaN	999.000	39	0	43	60	50	65	
Charmeleon	5	5	Fire	NaN	999.000	58	0	58	80	65	80	

In [174…
```
df.loc[(df['Type 2']=='Poison') & (df['Type 1'] == 'Grass'), 'Legendary'] = 'True'
df.head(5)
```

Out[174]:

Name	index	#	Type 1	Type 2	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generat
Bulbasaur	0	1	Grass	Poison	1.590	45	0	49	65	65	45	
Ivysaur	1	2	Grass	Poison	2.025	60	0	63	80	80	60	
Venusaur	2	3	Grass	Poison	2.625	80	0	83	100	100	80	
VenusaurMega Venusaur	3	3	Grass	Poison	3.125	80	0	123	122	120	80	
Charmander	4	4	Fire	NaN	999.000	39	0	43	60	50	65	

Modify multiple columns simultaneously.

It is also possible to change the value of multiple columns based on the first logical condition.

In [175…

```
df.loc[df['Total']>2,['Generation','Type 1']] = ['Best','Economist']
df.head(5)
```

Out[175]:

Name	index	#	Type 1	Type 2	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Ger
Bulbasaur	0	1	Grass	Poison	1.590	45	0	49	65	65	45	
Ivysaur	1	2	Economist	Poison	2.025	60	0	63	80	80	60	
Venusaur	2	3	Economist	Poison	2.625	80	0	83	100	100	80	
VenusaurMega Venusaur	3	3	Economist	Poison	3.125	80	0	123	122	120	80	
Charmander	4	4	Economist	NaN	999.000	39	0	43	60	50	65	

Conditional Changes using NumPy

We can also use NumPy methods to apply conditional changes to our `DataFrame`.

np.where()

It will allow us to change the elements that satisfy a particular condition, with the peculiarity that we can also change those elements that do not satisfy the condition. For example, suppose we want to change the *Defense* column elements, and instead of numbers just include 'good defense' and 'bad defense'.

In [8]:

```
df['Defense'] = np.where(df['Defense'] > 80, 'Good defense','Bad defense')
```

In [181…

```
df.head(5)
```

Out[181]:

Name	index	#	Type 1	Type 2	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Ger
Bulbasaur	0	1	Grass	Poison	1.590	45	0	Bad defense	65	65	45	
Ivysaur	1	2	Economist	Poison	2.025	60	0	Bad defense	80	80	60	
Venusaur	2	3	Economist	Poison	2.625	80	0	Good defense	100	100	80	
VenusaurMega Venusaur	3	3	Economist	Poison	3.125	80	0	Good defense	122	120	80	
Charmander	4	4	Economist	NaN	999.000	39	0	Bad defense	60	50	65	

np.select()

It expands the opportunities of `np.where()` since it allows to include more than one conditon. The syntaxt is `np.select(conditions,options)`. Let's see an example:

In [183…

```
conditions = [df['Type 1'] == 'Grass',df['Total'] > 2]
options = ['Red','White']

df['New Column'] = np.select(conditions,options)
df.head(4)
```

Out[183]:

Name	index	#	Type 1	Type 2	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Gener
Bulbasaur	0	1	Grass	Poison	1.590	45	0	Bad defense	65	65	45	
Ivysaur	1	2	Economist	Poison	2.025	60	0	Bad defense	80	80	60	
Venusaur	2	3	Economist	Poison	2.625	80	0	Good defense	100	100	80	
VenusaurMega Venusaur	3	3	Economist	Poison	3.125	80	0	Good defense	122	120	80	

Merge,Concat and Join

Pandas provides various facilities for easily combining together Series or DataFrame with various kinds of set logic for the indexes and relational algebra functionality in the case of join / merge-type operations. All this section is obtained from Pandas documentation, for more information visit their site.

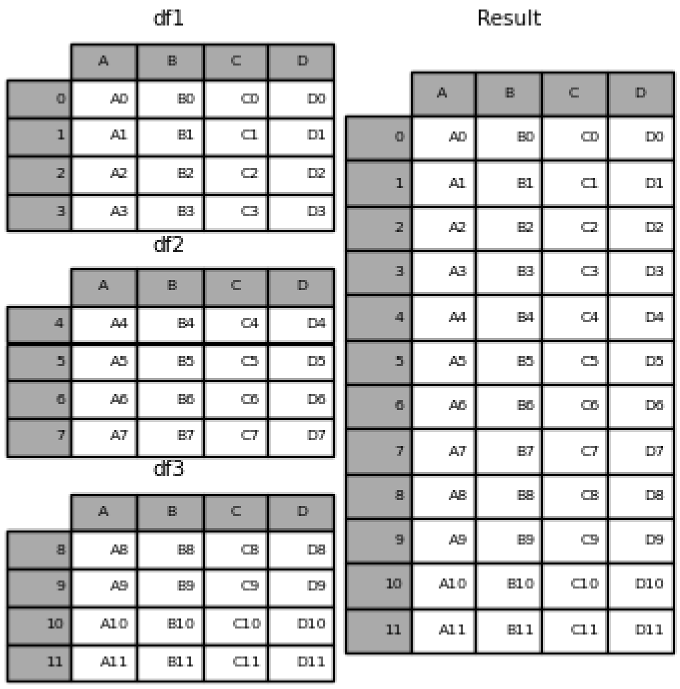
concat()

This is the function that we wil use whenever we have two or more `DataFrames` and we want to concatenate columns or rows from one `DataFrame` to another. Importantly, `concat()` allow us to concatenate in both axis, so we can either add more rows or more columns.

Let's first see a simple example:

In [ ]:
```
In [1]: df1 = pd.DataFrame(
   ...:     {
   ...:         "A": ["A0", "A1", "A2", "A3"],
   ...:         "B": ["B0", "B1", "B2", "B3"],
   ...:         "C": ["C0", "C1", "C2", "C3"],
   ...:         "D": ["D0", "D1", "D2", "D3"],
   ...:     },
   ...:     index=[0, 1, 2, 3],
   ...: )
   ...:

In [2]: df2 = pd.DataFrame(
   ...:     {
   ...:         "A": ["A4", "A5", "A6", "A7"],
   ...:         "B": ["B4", "B5", "B6", "B7"],
   ...:         "C": ["C4", "C5", "C6", "C7"],
   ...:         "D": ["D4", "D5", "D6", "D7"],
   ...:     },
   ...:     index=[4, 5, 6, 7],
   ...: )
   ...:

In [3]: df3 = pd.DataFrame(
   ...:     {
   ...:         "A": ["A8", "A9", "A10", "A11"],
   ...:         "B": ["B8", "B9", "B10", "B11"],
   ...:         "C": ["C8", "C9", "C10", "C11"],
   ...:         "D": ["D8", "D9", "D10", "D11"],
   ...:     },
   ...:     index=[8, 9, 10, 11],
   ...: )
   ...:

In [4]: frames = [df1, df2, df3]

In [5]: result = pd.concat(frames)
```

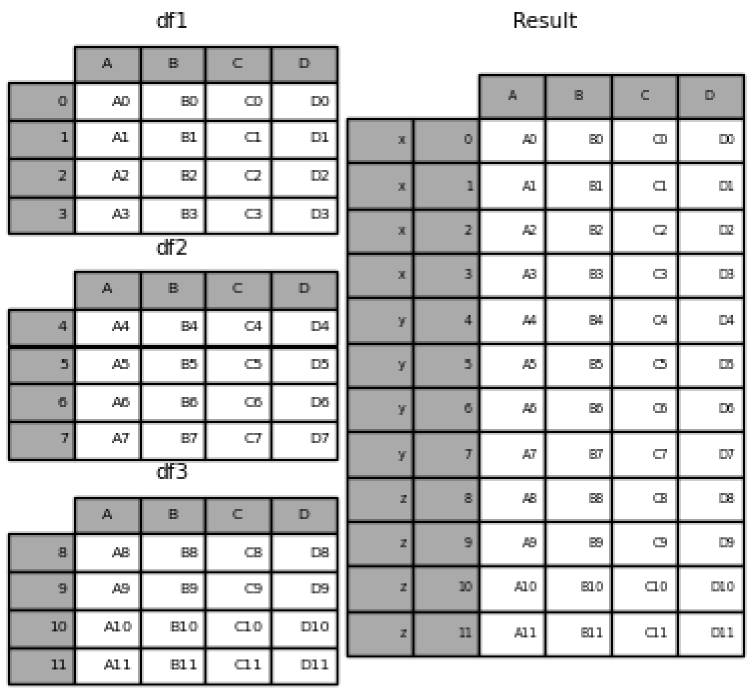
The result of this operation is:



Since all of the `DataFrames` have the same column names and different index values. Suppose we want to include an index to remember from which `DataFrame` each row comes from. We can do that using the `keys` argument:

In [185…

```
result = pd.concat(frames, keys=["x", "y", "z"])
```



Set logic on the other axes

Until now we have concatenated based on the rows. We can also concatenate based on the columns. Furthermore, we can specfy how to handle the axes that are not being concatenated. There are two options:

- **join = 'outer'** : This will take the union of all of them, resulting in zero infromation loss.

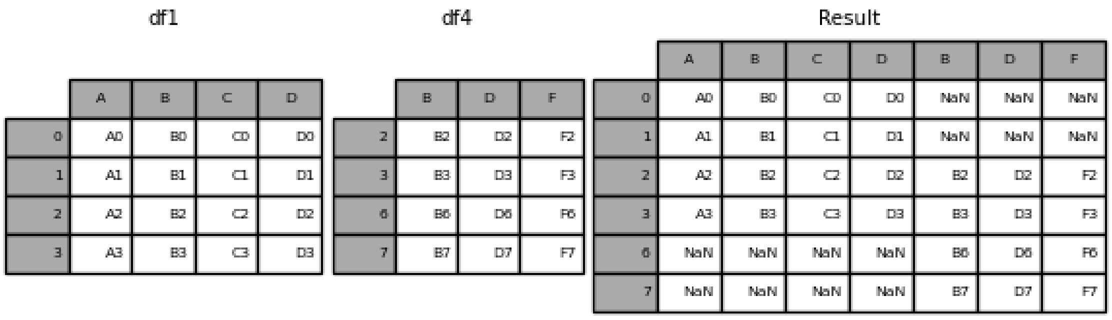
- **join = 'inner'** : This will take the intersection, with the information loss that this implies

Let's see an example:

In [187…

```
df4 = pd.DataFrame(
    {
        "B": ["B2", "B3", "B6", "B7"],
        "D": ["D2", "D3", "D6", "D7"],
        "F": ["F2", "F3", "F6", "F7"],
    },
    index=[2, 3, 6, 7],
)

result = pd.concat([df1, df4], axis=1, join ='outer')
print(result)
```

```
     A    B    C    D    B    D    F
0   A0   B0   C0   D0  NaN  NaN  NaN
1   A1   B1   C1   D1  NaN  NaN  NaN
2   A2   B2   C2   D2   B2   D2   F2
3   A3   B3   C3   D3   B3   D3   F3
6  NaN  NaN  NaN  NaN   B6   D6   F6
7  NaN  NaN  NaN  NaN   B7   D7   F7
```

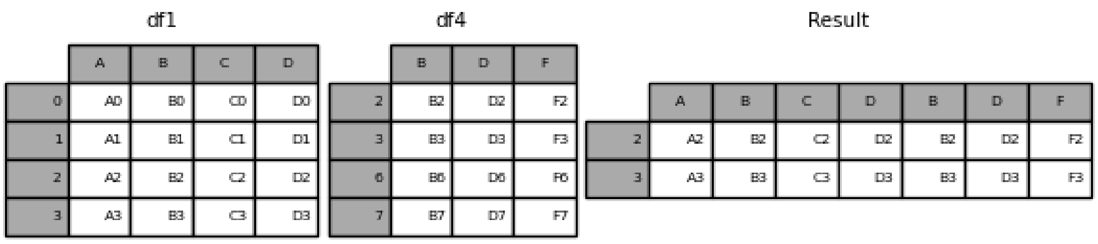


We can also do it with `inner`.

In [188…

```
result = pd.concat([df1, df4], axis=1, join="inner")
print(result)
```

```
    A   B   C   D   B   D   F
2  A2  B2  C2  D2  B2  D2  F2
3  A3  B3  C3  D3  B3  D3  F3
```



Notice that we are concatenating columns, and we are keeping or not rows based on wether they are common across `DataFrames`.

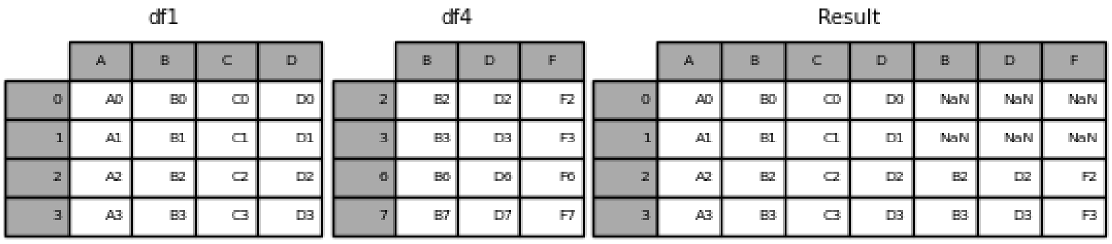
We can also rename the index from the second `DataFrame` as the first:

In [191…

```
pd.concat([df1, df4.reindex(df1.index)], axis=1)
```

Out[191]:

	A	B	C	D	B	D	F
0	A0	B0	C0	D0	NaN	NaN	NaN
1	A1	B1	C1	D1	NaN	NaN	NaN
2	A2	B2	C2	D2	B2	D2	F2
3	A3	B3	C3	D3	B3	D3	F3



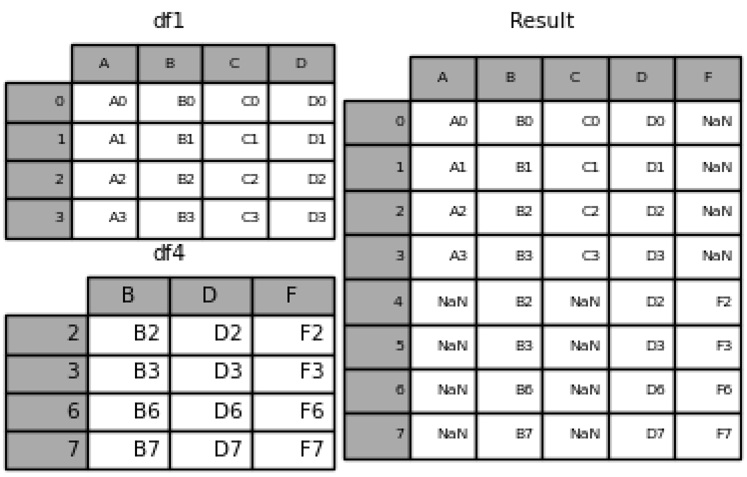
Ignoring indexes on the concatenation axis

In [193…

```
result = pd.concat([df1, df4], ignore_index=True, sort=False)
result.head(3)
```

Out[193]:

	A	B	C	D	F
0	A0	B0	C0	D0	NaN
1	A1	B1	C1	D1	NaN
2	A2	B2	C2	D2	NaN

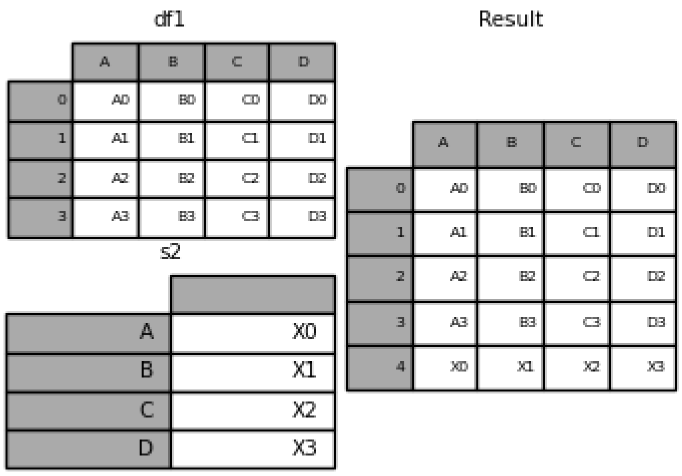


Appending rows to a DataFrame

We can also append a `Series` as a row to our `DataFrame`.

In [194…

```
s2 = pd.Series(["X0", "X1", "X2", "X3"], index=["A", "B", "C", "D"])

result = pd.concat([df1, s2.to_frame().T], ignore_index=True)
```



merge()

It is the most common way of including new columns to a `DataFrame` based in some standard conditions. It is the entry point for all standard database join operations. There are several cases to consider:

- **one-to-one** : when joininig two `DataFrame`objects on their indexes (which must contain unique values).
- **many-to-one** : when joining an index (unique) to one or more column is a different `DataFrame`.
- **many-to-many** : joining columns on columns.

The most important element doing the merge is the `key` which is the identifier on which we are going to merge. To specify which type of merge we are going to do we use the argument `how` . It specifies how to determine which keys are to be included in the resulting table. The options are:

- **left** : Use keys from left frame only.
- **right** : Uses keys from right frame only.
- **outer** : Use union of keys from both frames.
- **inner** : USe intersection of keys from both frames
- **cross** : Create the cartesian product of rows of both frames.

Let's see an example:

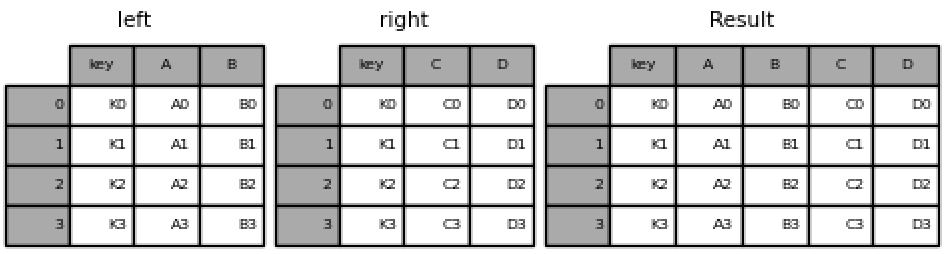
In [195…

```
left = pd.DataFrame(
    {
        "key": ["K0", "K1", "K2", "K3"],
        "A": ["A0", "A1", "A2", "A3"],
        "B": ["B0", "B1", "B2", "B3"],
    }
)

right = pd.DataFrame(
    {
        "key": ["K0", "K1", "K2", "K3"],
        "C": ["C0", "C1", "C2", "C3"],
        "D": ["D0", "D1", "D2", "D3"],
    }
```

```python
)

result = pd.merge(left, right, on="key")
result.head(2)
```

Out[195]:

	key	A	B	C	D
0	K0	A0	B0	C0	D0
1	K1	A1	B1	C1	D1



On the prevoius code we are merging `on="key`, which basically means `key` is our identifyer to do the merge. Furthermore, since we are not specifying the default is `how="inner"` which means we are using the intersection of the values of the variable `key`. Since in this case both `DataFrames` have the same values for the column `key`, this is trivial. We can also merge on two `keys`.

In [196…

```python
left = pd.DataFrame(
    {
        "key1": ["K0", "K0", "K1", "K2"],
        "key2": ["K0", "K1", "K0", "K1"],
        "A": ["A0", "A1", "A2", "A3"],
        "B": ["B0", "B1", "B2", "B3"],
    }
)

right = pd.DataFrame(
    {
        "key1": ["K0", "K1", "K1", "K2"],
        "key2": ["K0", "K0", "K0", "K0"],
        "C": ["C0", "C1", "C2", "C3"],
        "D": ["D0", "D1", "D2", "D3"],
    }
)

result = pd.merge(left, right, on=["key1", "key2"])
result.head(2)
```

Out[196]:

	key1	key2	A	B	C	D
0	K0	K0	A0	B0	C0	D0
1	K1	K0	A2	B2	C1	D1

left

	key1	key2	A	B
0	K0	K0	A0	B0
1	K0	K1	A1	B1
2	K1	K0	A2	B2
3	K2	K1	A3	B3

right

	key1	key2	C	D
0	K0	K0	C0	D0
1	K1	K0	C1	D1
2	K1	K0	C2	D2
3	K2	K0	C3	D3

Result

	key1	key2	A	B	C	D
0	K0	K0	A0	B0	C0	D0
1	K1	K0	A2	B2	C1	D1
2	K1	K0	A2	B2	C2	D2

Notice since by default we are using `how = "inner"` we are only taking the combinations of `key1` and `key2` that are common across `DataFrames`. The others are being dropped.

Left merge

It merges on the same index across `DataFrames` which is specified by `on="XXX"` but it only considers the values of the index that appear on the `left` `DataFrame`.

In [197…

```
result = pd.merge(left, right, how="left", on=["key1", "key2"])
```

left

	key1	key2	A	B
0	K0	K0	A0	B0
1	K0	K1	A1	B1
2	K1	K0	A2	B2
3	K2	K1	A3	B3

right

	key1	key2	C	D
0	K0	K0	C0	D0
1	K1	K0	C1	D1
2	K1	K0	C2	D2
3	K2	K0	C3	D3

Result

	key1	key2	A	B	C	D
0	K0	K0	A0	B0	C0	D0
1	K0	K1	A1	B1	NaN	NaN
2	K1	K0	A2	B2	C1	D1
3	K1	K0	A2	B2	C2	D2
4	K2	K1	A3	B3	NaN	NaN

Right merge

In [198…

```
result = pd.merge(left, right, how="right", on=["key1", "key2"])
```

left

	key1	key2	A	B
0	K0	K0	A0	B0
1	K0	K1	A1	B1
2	K1	K0	A2	B2
3	K2	K1	A3	B3

right

	key1	key2	C	D
0	K0	K0	C0	D0
1	K1	K0	C1	D1
2	K1	K0	C2	D2
3	K2	K0	C3	D3

Result

	key1	key2	A	B	C	D
0	K0	K0	A0	B0	C0	D0
1	K1	K0	A2	B2	C1	D1
2	K1	K0	A2	B2	C2	D2
3	K2	K0	NaN	NaN	C3	D3

Outer merge

In [199…

```
result = pd.merge(left, right, how="outer", on=["key1", "key2"])
```

left

	key1	key2	A	B
0	K0	K0	A0	B0
1	K0	K1	A1	B1
2	K1	K0	A2	B2
3	K2	K1	A3	B3

right

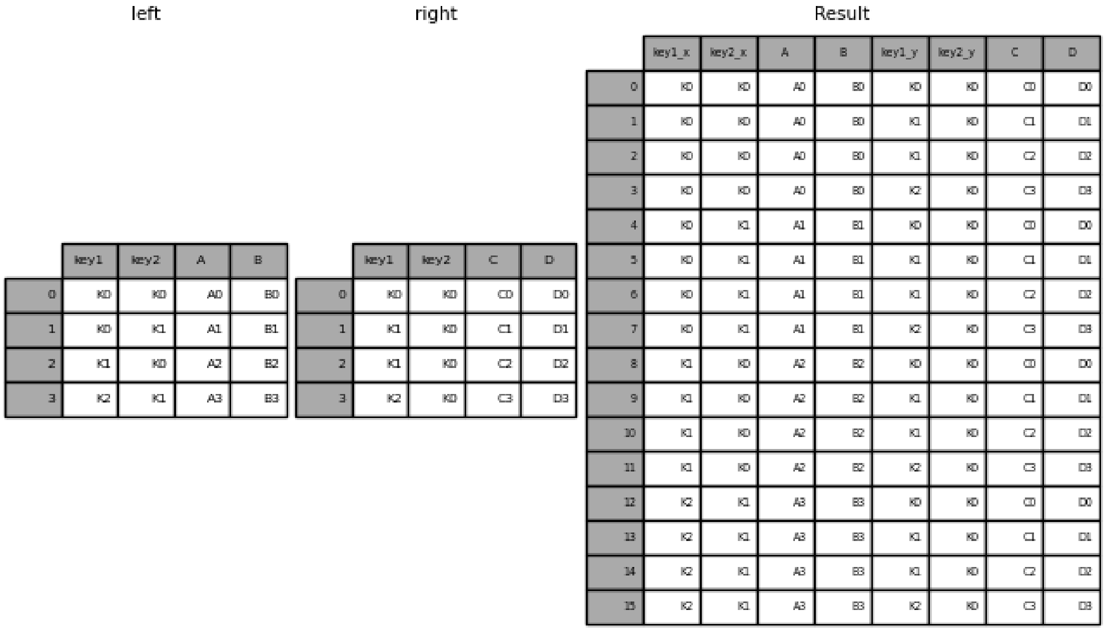
	key1	key2	C	D
0	K0	K0	C0	D0
1	K1	K0	C1	D1
2	K1	K0	C2	D2
3	K2	K0	C3	D3

Result

	key1	key2	A	B	C	D
0	K0	K0	A0	B0	C0	D0
1	K0	K1	A1	B1	NaN	NaN
2	K1	K0	A2	B2	C1	D1
3	K1	K0	A2	B2	C2	D2
4	K2	K1	A3	B3	NaN	NaN
5	K2	K0	NaN	NaN	C3	D3

Cross merge

```
In [200…
result = pd.merge(left, right, how="cross")
```



Checking for duplicate keys

Users can use the `validate` argument to automatically check whether there are unexpected duplicates in their merge keys. Checking key uniqueness is also a good way to ensure user data structures are as expected.

```
In [9]:
left = pd.DataFrame({"A": [1, 2], "B": [1, 2]})
right = pd.DataFrame({"A": [4, 5, 6], "B": [2, 2, 2]})

result = pd.merge(left, right, on="B", how="outer", validate="one_to_one")
```

```
---------------------------------------------------------------------------
MergeError                                Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_18972\2149159108.py in <module>
      2 right = pd.DataFrame({"A": [4, 5, 6], "B": [2, 2, 2]})
      3 
----> 4 result = pd.merge(left, right, on="B", how="outer", validate="one_to_one")

~\anaconda3\lib\site-packages\pandas\core\reshape\merge.py in merge(left, right, how, on, left_on, right_on, left_index, right_index, sort, suffixes, copy, indicator, validate)
    105     validate: str | None = None,
    106 ) -> DataFrame:
--> 107     op = _MergeOperation(
    108         left,
    109         right,

~\anaconda3\lib\site-packages\pandas\core\reshape\merge.py in __init__(self, left, right, how, on, left_on, right_on, axis, left_index, right_index, sort, suffixes, copy, indicator, validate)
    708         # are in fact unique.
    709         if validate is not None:
--> 710             self._validate(validate)
    711 
    712     def get_result(self) -> DataFrame:

~\anaconda3\lib\site-packages\pandas\core\reshape\merge.py in _validate(self, validate)
   1434                 )
   1435             elif not right_unique:
-> 1436                 raise MergeError(
   1437                     "Merge keys are not unique in right dataset; not a one-to-one merge"
   1438                 )

MergeError: Merge keys are not unique in right dataset; not a one-to-one merge
```

If the user is aware of the duplicates in the right DataFrame but wants to ensure there are no duplicates in the left DataFrame, one can use the `validate='one_to_many'` argument instead, which will not raise an exception.

In [202…

```python
pd.merge(left, right, on="B", how="outer", validate="one_to_many")
```

Out[202]:

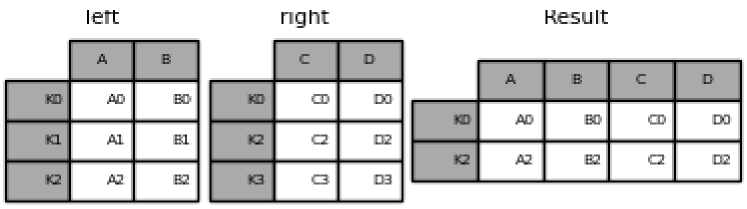
	A_x	B	A_y
0	1	1	NaN
1	2	2	4.0
2	2	2	5.0
3	2	2	6.0

Joining on index

For this we will use the method `join()`. It is a conviniemt method for combining the columns of two potentially differently-indexed `DataFrames`. It is the same procedure as with `merge()` with the peculiarity that the `on='XXX'` is now the index of the `DataFrame`.

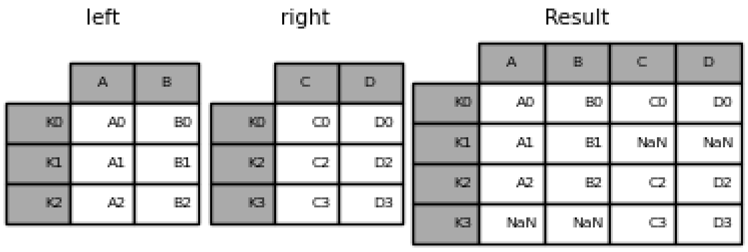
In [203…

```python
left = pd.DataFrame(
    {"A": ["A0", "A1", "A2"], "B": ["B0", "B1", "B2"]}, index=["K0", "K1", "K2"]
)

right = pd.DataFrame(
    {"C": ["C0", "C2", "C3"], "D": ["D0", "D2", "D3"]}, index=["K0", "K2", "K3"]
)

result = left.join(right)
```



In [204…

```python
result = left.join(right, how="outer")
```

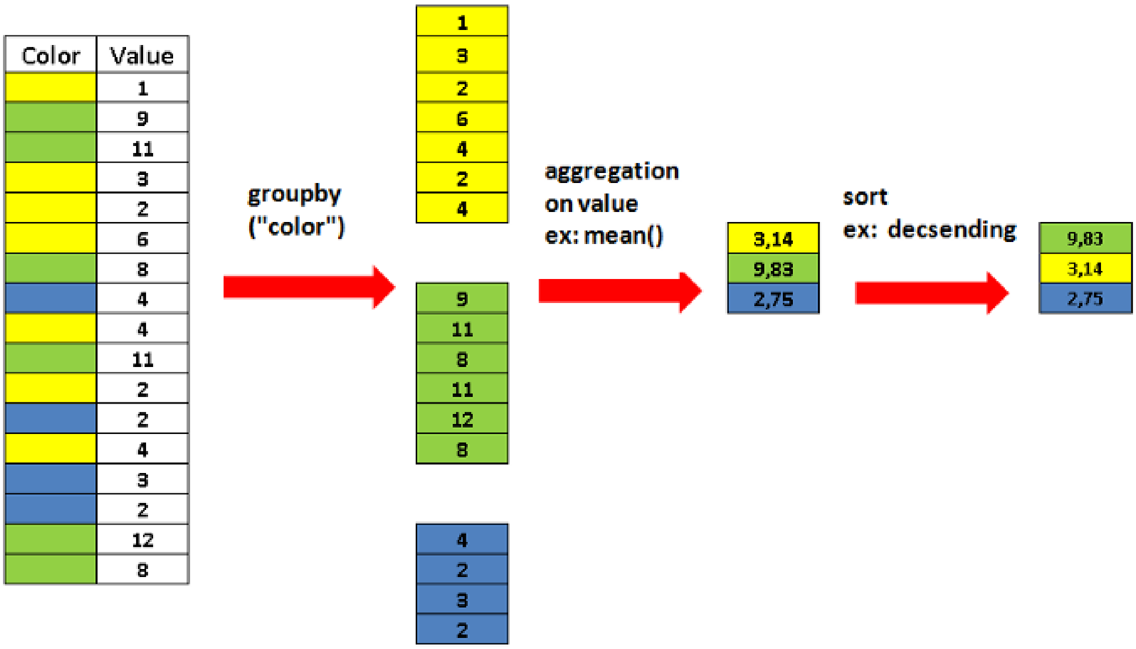


Groupby

One of the key features of Pandas is the `groupby` method. It allows to aggregate, combine and transform the data. We then refer to `groupby` as a process involving one or more of the following steps:

- **splitting** data into groups based on some criteria
- **applying** a function to each group independently.
- **combining** the results into a data structure.

The main idea behind is that we can combine elements by groups, and apply a function to the elements. For example, obtain the mean strength of each Pokemon. For an extensive revision visit this site.



In [206…

```
df = pd.read_csv('Pokemon.csv')
```

Splitting objects into groups

It returns a list of tuples of each unique value on the `'Type 1'` column with the sliced `DataFrame`.

In [218…

```
a = df.groupby(['Type 1'])
print(a.groups)
for i in a :
    print(i[1].head(3))
```

{'Bug': [13, 14, 15, 16, 17, 18, 19, 51, 52, 53, 54, 132, 136, 137, 179, 180, 181, 182, 208, 219, 220, 228, 229, 230, 231, 232, 288, 289, 290, 291, 292, 307, 308, 314, 315, 316, 342, 343, 446, 447, 457, 458, 459, 460, 461, 462, 463, 520, 600, 601, 602, 603, 604, 605, 618, 619, 649, 650, 656, 657, 677, 678, 693, 697, 698, 717, 732, 733, 734], 'Dark': [212, 213, 233, 246, 247, 248, 284, 285, 326, 327, 392, 393, 478, 512, 549, 568, 569, 620, 621, 631, 632, 685, 686, 690, 691, 694, 695, 696, 756, 757, 793], 'Dragon': [159, 160, 161, 365, 366, 406, 407, 408, 409, 417, 418, 419, 420, 425, 426, 491, 492, 493, 494, 671, 672, 673, 682, 706, 707, 710, 711, 712, 774, 775, 776, 794], 'Electric': [30, 31, 88, 89, 108, 109, 134, 146, 157, 186, 193, 194, 195, 196, 258, 262, 337, 338, 339, 340, 341, 448, 449, 450, 464, 513, 517, 531, 532, 533, 534, 535, 536, 581, 582, 648, 663, 664, 665, 704, 705, 764, 765, 772], 'Fairy': [40, 41, 187, 189, 190, 225, 226, 519, 737, 738, 739, 752, 753, 754, 755, 770, 792], 'Fighting': [61, 62, 72, 73, 74, 114, 115, 255, 256, 320, 321, 334, 335, 336, 496, 497, 498, 592, 593, 594, 598, 599, 680, 681, 742, 743, 771], 'Fire': [4, 5, 6, 7, 8, 42, 43, 63, 64, 83, 84, 135, 147, 158, 169, 170, 171, 236, 237, 259, 263, 270, 276, 277, 278, 279, 352, 353, 354, 355, 435, 436, 437, 518, 542, 557, 558, 559, 572, 573, 614, 615, 616, 692, 721, 722, 723, 730, 731, 735, 736, 799], 'Flying': [702, 703, 790, 791], 'Ghost': [99, 100, 101, 102, 215, 385, 386, 387, 388, 389, 472, 473, 477, 490, 529, 544, 545, 623, 624, 668, 669, 670, 778, 779, 780, 781, 782, 783, 784, 785, 786, 787], 'Grass': [0, 1, 2, 3, 48, 49, 50, 75, 76, 77, 110, 111, 122, 166, 167, 168, 197, 202, 203, 204, 206, 207, 272, 273, 274, 275, 296, 297, 298, 309, 310, 344, 362, 363, 390, 432, 433, 434, 451, 452, 467, 468, 505, 509, 510, 511, 516, 521, 550, 551, 554, 555, 556, 570, 571, 606, 607, 608, 609, 617, 651, 652, 658, 659, 701, 718, 719, 720, 740, 741], 'Ground': [32, 33, 55, 56, 112, 113, 119, 120, 222, 250, 251, 359, 360, 361, 375, 376, 423, 424, 499, 500, 515, 523, 588, 589, 611, 612, 613, 679, 683, 684, 708, 709], 'Ice': [133, 156, 238, 239, 243, 257, 395, 396, 397, 398, 399, 400, 415, 522, 524, 530, 643, 644, 645, 674, 675, 676, 788, 789], 'Normal': [20, 21, 22, 23, 24, 25, 26, 27, 44, 45, 57, 58, 90, 91, 92, 116, 121, 123, 124, 138, 143, 144, 148, 155, 175, 176, 177, 178, 188, 205, 218, 221, 234, 235, 252, 253, 254, 260, 261, 286, 287, 299, 300, 311, 312, 313, 317, 318, 319, 322, 324, 325, 358, 364, 367, 383, 384, 441, 442, 443, 444, 445, 471, 474, 475, 476, 479, 480, 488, 489, 495, 514, 525, 543, 552, 563, 564, 565, 566, 567, 578, 579, 580, 590, 591, 633, 634, 646, 647, 687, 688, 689, 715, 716, 727, 728, 729, 744], 'Poison': [28, 29, 34, 35, 36, 37, 38, 39, 46, 47, 95, 96, 117, 118, 183, 345, 346, 368, 482, 483, 501, 502, 503, 504, 629, 630, 760, 761], 'Psychic': [68, 69, 70, 71, 104, 105, 131, 162, 163, 164, 165, 191, 192, 211, 216, 217, 269, 271, 303, 304, 305, 306, 356, 357, 391, 394, 428, 429, 430, 431, 481, 487, 526, 527, 537, 538, 539, 546, 553, 576, 577, 586, 587, 622, 635, 636, 637, 638, 639, 640, 666, 667, 745, 746, 747, 797, 798], 'Rock': [80, 81, 82, 103, 149, 150, 151, 152, 153, 154, 200, 265, 266, 267, 268, 323, 369, 370, 377, 378, 379, 380, 414, 453, 454, 455, 456, 486, 528, 583, 584, 585, 627, 628, 700, 758, 759, 766, 767, 768, 769, 773, 795, 796], 'Steel': [223, 224, 245, 328, 329, 330, 331, 332, 333, 410, 411, 412, 413, 416, 427, 484, 485, 540, 660, 661, 662, 699, 748, 749, 750, 751, 777], 'Water': [9, 10, 11, 12, 59, 60, 65, 66, 67, 78, 79, 85, 86, 87, 93, 94, 97, 98, 106, 107, 125, 126, 127, 128, 129, 130, 139, 140, 141, 142, 145, 172, 173, 174, 184, 185, 198, 199, 201, 209, 210, 214, 227, 240, 241, 242, 244, 249, 264, 280, 281, 282, 283, 293, 294, 295, 301, 302, 347, 348, 349, 350, 351, 371, 372, 373, 374, 381, 382, 401, 402, 403, 404, 405, 421, 422, 438, 439, 440, 465, 466, 469, 470, 506, 507, 508, 541, 547, 548, 560, 561, 562, 574, 575, 595, 596, 597, 610, 625, 626, ...]}

```
     #        Name Type 1  Type 2  Total  HP  Attack  Defense  Sp. Atk  \
13  10    Caterpie    Bug     NaN    195  45      30       35       20
14  11     Metapod    Bug     NaN    205  50      20       55       25
15  12  Butterfree    Bug  Flying    395  60      45       50       90

    Sp. Def  Speed  Generation  Legendary
13       20     45           1      False
14       25     30           1      False
15       80     70           1      False
       #     Name Type 1  Type 2  Total  HP  Attack  Defense  Sp. Atk  \
212  197  Umbreon   Dark     NaN    525  95      65      110       60
213  198  Murkrow   Dark  Flying    405  60      85       42       85
233  215  Sneasel   Dark     Ice    430  55      95       55       35
```

```
     Sp. Def  Speed  Generation  Legendary
212      130     65           2      False
213       42     91           2      False
233       75    115           2      False
       #       Name  Type 1  Type 2  Total  HP  Attack  Defense  Sp. Atk  \
159  147    Dratini  Dragon     NaN    300  41      64       45       50
160  148  Dragonair  Dragon     NaN    420  61      84       65       70
161  149  Dragonite  Dragon  Flying    600  91     134       95      100

     Sp. Def  Speed  Generation  Legendary
159       50     50           1      False
160       70     70           1      False
161      100     80           1      False
     #       Name    Type 1 Type 2  Total  HP  Attack  Defense  Sp. Atk  \
30  25    Pikachu  Electric    NaN    320  35      55       40       50
31  26     Raichu  Electric    NaN    485  60      90       55       90
88  81  Magnemite  Electric  Steel    325  25      35       70       95

    Sp. Def  Speed  Generation  Legendary
30       50     90           1      False
31       80    110           1      False
88       55     45           1      False
       #      Name Type 1 Type 2  Total  HP  Attack  Defense  Sp. Atk  \
40    35  Clefairy  Fairy    NaN    323  70      45       48       60
41    36  Clefable  Fairy    NaN    483  95      70       73       95
187  173    Cleffa  Fairy    NaN    218  50      25       28       45

     Sp. Def  Speed  Generation  Legendary
40        65     35           1      False
41        90     60           1      False
187       55     15           2      False
     #      Name    Type 1 Type 2  Total  HP  Attack  Defense  Sp. Atk  \
61  56    Mankey  Fighting    NaN    305  40      80       35       35
62  57  Primeape  Fighting    NaN    455  65     105       60       60
72  66    Machop  Fighting    NaN    305  70      80       50       35

    Sp. Def  Speed  Generation  Legendary
61       45     70           1      False
62       70     95           1      False
72       35     35           1      False
   #        Name Type 1  Type 2  Total  HP  Attack  Defense  Sp. Atk  Sp. Def  \
4  4  Charmander   Fire     NaN    309  39      52       43       60       50
5  5  Charmeleon   Fire     NaN    405  58      64       58       80       65
6  6   Charizard   Fire  Flying    534  78      84       78      109       85

   Speed  Generation  Legendary
4     65           1      False
5     80           1      False
6    100           1      False
       #                     Name  Type 1  Type 2  Total  HP  Attack  Defense  \
702  641  TornadusIncarnate Forme  Flying     NaN    580  79     115       70
703  641    TornadusTherian Forme  Flying     NaN    580  79     100       80
790  714                   Noibat  Flying  Dragon    245  40      30       35

     Sp. Atk  Sp. Def  Speed  Generation  Legendary
702      125       80    111           5       True
703      110       90    121           5       True
790       45       40     55           6      False
       #     Name Type 1  Type 2  Total  HP  Attack  Defense  Sp. Atk  Sp. Def  \
99   92   Gastly  Ghost  Poison    310  30      35       30      100       35
100  93  Haunter  Ghost  Poison    405  45      50       45      115       55
101  94   Gengar  Ghost  Poison    500  60      65       60      130       75

     Speed  Generation  Legendary
```

```
99      80           1      False
100     95           1      False
101    110           1      False
   #       Name Type 1  Type 2  Total  HP  Attack  Defense  Sp. Atk  Sp. Def  \
0  1  Bulbasaur  Grass  Poison    318  45      49       49       65       65
1  2    Ivysaur  Grass  Poison    405  60      62       63       80       80
2  3   Venusaur  Grass  Poison    525  80      82       83      100      100

   Speed  Generation  Legendary
0     45           1      False
1     60           1      False
2     80           1      False
     #       Name  Type 1 Type 2  Total  HP  Attack  Defense  Sp. Atk  \
32  27  Sandshrew  Ground    NaN    300  50      75       85       20
33  28  Sandslash  Ground    NaN    450  75     100      110       45
55  50    Diglett  Ground    NaN    265  10      55       25       35

    Sp. Def  Speed  Generation  Legendary
32       30     40           1      False
33       55     65           1      False
55       45     95           1      False
       #      Name Type 1   Type 2  Total  HP  Attack  Defense  Sp. Atk  \
133  124      Jynx    Ice  Psychic    455  65      50       35      115
156  144  Articuno    Ice   Flying    580  90      85      100       95
238  220    Swinub    Ice   Ground    250  50      50       40       30

     Sp. Def  Speed  Generation  Legendary
133       95     95           1      False
156      125     85           1       True
238       30     50           2      False
     #       Name  Type 1  Type 2  Total  HP  Attack  Defense  Sp. Atk  \
20  16     Pidgey  Normal  Flying    251  40      45       40       35
21  17  Pidgeotto  Normal  Flying    349  63      60       55       50
22  18    Pidgeot  Normal  Flying    479  83      80       75       70

    Sp. Def  Speed  Generation  Legendary
20       35     56           1      False
21       50     71           1      False
22       70    101           1      False
     #        Name  Type 1 Type 2  Total  HP  Attack  Defense  Sp. Atk  \
28  23       Ekans  Poison    NaN    288  35      60       44       40
29  24       Arbok  Poison    NaN    438  60      85       69       65
34  29  Nidoran(F)  Poison    NaN    275  55      47       52       40

    Sp. Def  Speed  Generation  Legendary
28       54     55           1      False
29       79     80           1      False
34       40     41           1      False
     #      Name   Type 1 Type 2  Total  HP  Attack  Defense  Sp. Atk  \
68  63      Abra  Psychic    NaN    310  25      20       15      105
69  64   Kadabra  Psychic    NaN    400  40      35       30      120
70  65  Alakazam  Psychic    NaN    500  55      50       45      135

    Sp. Def  Speed  Generation  Legendary
68       55     90           1      False
69       70    105           1      False
70       95    120           1      False
     #      Name Type 1  Type 2  Total  HP  Attack  Defense  Sp. Atk  Sp. Def  \
80  74   Geodude   Rock  Ground    300  40      80      100       30       30
81  75  Graveler   Rock  Ground    390  55      95      115       45       45
82  76     Golem   Rock  Ground    495  80     120      130       55       65

    Speed  Generation  Legendary
80     20           1      False
```

```
81      35              1       False
82      45              1       False
        #                   Name  Type 1  Type 2  Total  HP  Attack  Defense  \
223   208                Steelix   Steel  Ground    510  75      85      200
224   208  SteelixMega Steelix   Steel  Ground    610  75     125      230
245   227               Skarmory   Steel  Flying    465  65      80      140

     Sp. Atk  Sp. Def  Speed  Generation  Legendary
223       55       65     30           2      False
224       55       95     30           2      False
245       40       70     70           2      False
    #       Name Type 1 Type 2  Total  HP  Attack  Defense  Sp. Atk  Sp. Def  \
9   7   Squirtle  Water    NaN    314  44      48       65       50       64
10  8  Wartortle  Water    NaN    405  59      63       80       65       80
11  9  Blastoise  Water    NaN    530  79      83      100       85      105

    Speed  Generation  Legendary
9      43           1      False
10     58           1      False
11     78           1      False
```

Create a new DataFrame

The most common use is when we just care about the aggregate characteristics of some groups. Similar to Stata `collapse`. The following code provides the mean of all the variables by `Type 1`.

In [220…

```
dfmean = df.groupby(['Type 1']).mean()
dfmean.head(4)
```

Out[220]:

Type 1	#	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speec
Bug	334.492754	378.927536	56.884058	70.971014	70.724638	53.869565	64.797101	61.68115
Dark	461.354839	445.741935	66.806452	88.387097	70.225806	74.645161	69.516129	76.16129
Dragon	474.375000	550.531250	83.312500	112.125000	86.375000	96.843750	88.843750	83.03125
Electric	363.500000	443.409091	59.795455	69.090909	66.295455	90.022727	73.704545	84.50000

Similarly, we can also sum all the values:

In [221…

```
dfsum  = df.groupby(['Type 1']).sum()
dfsum.head(2)
```

Out[221]:

Type 1	#	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Legendary
Bug	23080	26146	3925	4897	4880	3717	4471	4256	222	0
Dark	14302	13818	2071	2740	2177	2314	2155	2361	125	2

Suppose we are only interested into one of the values of the `DataFrame`. For example, the average *Attack*. We could do:

In [223…

```
df.groupby(['Type 1'])['Attack'].mean().head(4)
```

Out[223]:

```
Type 1
Bug          70.971014
Dark         88.387097
Dragon      112.125000
Electric     69.090909
Name: Attack, dtype: float64
```

Gropuby with multiple columns

We can also do a groupby with multiple columns.

In [226…

```
df.groupby(['Type 1','Type 2'])['Attack'].mean().head(20)
```

Out[226]:

```
Type 1  Type 2
Bug     Electric     62.000000
        Fighting    155.000000
        Fire         72.500000
        Flying       70.142857
        Ghost        90.000000
        Grass        73.833333
        Ground       62.000000
        Poison       68.333333
        Rock         56.666667
        Steel       114.714286
        Water        30.000000
Dark    Dragon       85.000000
        Fighting     82.500000
        Fire         80.000000
        Flying       92.200000
        Ghost        80.000000
        Ice         107.500000
        Psychic      73.000000
        Steel       105.000000
Dragon  Electric    150.000000
Name: Attack, dtype: float64
```

It is important to mention that the index of the result dataframe are the columns that we are grouping by:

In [227…

```
a = df.groupby(['Type 1','Type 2'])['Attack'].mean()
a.index
```

Out[227]:
```
MultiIndex([(  'Bug', 'Electric'),
            (  'Bug', 'Fighting'),
            (  'Bug',     'Fire'),
            (  'Bug',   'Flying'),
            (  'Bug',    'Ghost'),
            (  'Bug',    'Grass'),
            (  'Bug',   'Ground'),
            (  'Bug',   'Poison'),
            (  'Bug',     'Rock'),
            (  'Bug',    'Steel'),
            ...
            ('Water', 'Fighting'),
            ('Water',   'Flying'),
            ('Water',    'Ghost'),
            ('Water',    'Grass'),
            ('Water',   'Ground'),
            ('Water',      'Ice'),
            ('Water',   'Poison'),
            ('Water',  'Psychic'),
            ('Water',     'Rock'),
            ('Water',    'Steel')],
           names=['Type 1', 'Type 2'], length=136)
```

Aggregate multiple statistics

Until now we have just been able to obtain one new column returning the mean or the sum of the grup. However, we can obtain more than one statistic with just one groupby using the `agg()` method.

In [228…
```
df.groupby(['Type 1'])['Attack'].agg(['mean','sum']).head(5)
```

Out[228]:

Type 1	mean	sum
Bug	70.971014	4897
Dark	88.387097	2740
Dragon	112.125000	3588
Electric	69.090909	3040
Fairy	61.529412	1046

In [229…
```
# we can put the name that we want:
df.groupby(['Type 1'])['Attack'].agg(Average = ('mean'),Totalsum = ('sum')).head(5
```

Out[229]:

Type 1	Average	Totalsum
Bug	70.971014	4897
Dark	88.387097	2740
Dragon	112.125000	3588
Electric	69.090909	3040
Fairy	61.529412	1046

Apply

Pandas apply allows to apply a function along an axis of the `DataFrame`. We can apply built-in functions, our functions and lambda functions. It improves the efficiency of our code with an elegant syntaxt.

In [232…
```
# Modify the attack
df['Attack'].apply(lambda x: x*2 +1)
```

Out[232]:
```
0       99
1      125
2      165
3      201
4      105
      ...
795    201
796    321
797    221
798    321
799    221
Name: Attack, Length: 800, dtype: int64
```

In [233…
```
# Create a function

def modify(x):

    return x*2+1

df['Attack'].apply(modify)
```

Out[233]:
```
0       99
1      125
2      165
3      201
4      105
      ...
795    201
796    321
797    221
798    321
799    221
Name: Attack, Length: 800, dtype: int64
```

In [ ]:
```

```