DATA SCIENTIST

In this tutorial, I only explain you what you need to be a data scientist neither more nor less.

Data scientist need to have these skills:

1. Basic Tools: Like python, R or SQL. You do not need to know everything. What you only need is to learn how to use **python**
2. Basic Statistics: Like mean, median or standart deviation. If you know basic statistics, you can use **python** easily.
3. Data Munging: Working with messy and difficult data. Like a inconsistent date and string formatting. As you guess, **python** helps us.
4. Data Visualization: Title is actually explanatory. We will visualize the data with **python** like matplot and seaborn libraries.
5. Machine Learning: You do not need to understand math behind the machine learning technique. You only need is understanding basics of machine learning and learning how to implement it while using **python**.

As a summary we will learn python to be data scientist !!!

Content:

10. Statistic
    1. https://www.kaggle.com/kanncaa1/basic-statistic-tutorial-for-beginners
11. Deep Learning with Pytorch
    1. Artificial Neural Network: https://www.kaggle.com/kanncaa1/pytorch-tutorial-for-deep-learning-lovers
    2. Convolutional Neural Network: https://www.kaggle.com/kanncaa1/pytorch-tutorial-for-deep-learning-lovers
    3. Recurrent Neural Network: https://www.kaggle.com/kanncaa1/recurrent-neural-network-with-pytorch

```
# This Python 3 environment comes with many helpful analytics libraries installed
# It is defined by the kaggle/python docker image: https://github.com/kaggle/docker-python
# For example, here's several helpful packages to load in

import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import matplotlib.pyplot as plt
import seaborn as sns  # visualization tool

# Input data files are available in the "../input/" directory.
# For example, running this (by clicking run or pressing Shift+Enter) will list the files in the input
directory

from subprocess import check_output
# print(check_output(["ls", "../input"]).decode("utf8"))

# Any results you write to the current directory are saved as output.
```

```
data = pd.read_csv('pokemon.csv')
```

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 800 entries, 0 to 799
Data columns (total 12 columns):
 #   Column      Non-Null Count  Dtype 
---  ------      --------------  ----- 
 0   #           800 non-null    int64 
 1   Name        799 non-null    object
 2   Type 1      800 non-null    object
 3   Type 2      414 non-null    object
 4   HP          800 non-null    int64 
 5   Attack      800 non-null    int64 
 6   Defense     800 non-null    int64 
 7   Sp. Atk     800 non-null    int64 
 8   Sp. Def     800 non-null    int64 
 9   Speed       800 non-null    int64 
 10  Generation  800 non-null    int64 
 11  Legendary   800 non-null    bool  
dtypes: bool(1), int64(8), object(3)
memory usage: 69.7+ KB
```

```
data.corr()
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	#	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Legendary
#	1.000000	0.097712	0.102664	0.094691	0.089199	0.085596	0.012181	0.983428	0.154336
HP	0.097712	1.000000	0.422386	0.239622	0.362380	0.378718	0.175952	0.058683	0.273620
Attack	0.102664	0.422386	1.000000	0.438687	0.396362	0.263990	0.381240	0.051451	0.345408
Defense	0.094691	0.239622	0.438687	1.000000	0.223549	0.510747	0.015227	0.042419	0.246377
Sp. Atk	0.089199	0.362380	0.396362	0.223549	1.000000	0.506121	0.473018	0.036437	0.448907
Sp. Def	0.085596	0.378718	0.263990	0.510747	0.506121	1.000000	0.259133	0.028486	0.363937
Speed	0.012181	0.175952	0.381240	0.015227	0.473018	0.259133	1.000000	-0.023121	0.326715
Generation	0.983428	0.058683	0.051451	0.042419	0.036437	0.028486	-0.023121	1.000000	0.079794
Legendary	0.154336	0.273620	0.345408	0.246377	0.448907	0.363937	0.326715	0.079794	1.000000

```python
#correlation map
f,ax = plt.subplots(figsize=(18, 18))
sns.heatmap(data.corr(), annot=True, linewidths=.5, fmt= '.1f',ax=ax)
plt.show()
```

png

```python
data.head(10)
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	#	Name	Type 1	Type 2	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Legendary
0	1	Bulbasaur	Grass	Poison	45	49	49	65	65	45	1	False
1	2	Ivysaur	Grass	Poison	60	62	63	80	80	60	1	False
2	3	Venusaur	Grass	Poison	80	82	83	100	100	80	1	False
3	4	Mega Venusaur	Grass	Poison	80	100	123	122	120	80	1	False
4	5	Charmander	Fire	NaN	39	52	43	60	50	65	1	False
5	6	Charmeleon	Fire	NaN	58	64	58	80	65	80	1	False
6	7	Charizard	Fire	Flying	78	84	78	109	85	100	1	False
7	8	Mega Charizard X	Fire	Dragon	78	130	111	130	85	100	1	False
8	9	Mega Charizard Y	Fire	Flying	78	104	78	159	115	100	1	False
9	10	Squirtle	Water	NaN	44	48	65	50	64	43	1	False

```python
data.columns
```

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

1. INTRODUCTION TO PYTHON

MATPLOTLIB

Matplot is a python library that help us to plot data. The easiest and basic plots are line, scatter and histogram plots.

- Line plot is better when x axis is time.
- Scatter is better when there is correlation between two variables
- Histogram is better when we need to see distribution of numerical data.
- Customization: Colors,labels,thickness of line, title, opacity, grid, figsize, ticks of axis and linestyle

```
# Line Plot
# color = color, label = label, linewidth = width of line, alpha = opacity, grid = grid, linestyle = sytle of line
data.Speed.plot(kind = 'line', color = 'g',label = 'Speed',linewidth=1,alpha = 0.5,grid = True,linestyle = ':')
data.Defense.plot(color = 'r',label = 'Defense',linewidth=1, alpha = 0.5,grid = True,linestyle = '-.')
plt.legend(loc='upper right')     # legend = puts label into plot
plt.xlabel('x axis')              # label = name of label
plt.ylabel('y axis')
plt.title('Line Plot')            # title = title of plot
plt.show()
```

png

```
# Scatter Plot
# x = attack, y = defense
data.plot(kind='scatter', x='Attack', y='Defense',alpha = 0.5,color = 'red')
plt.xlabel('Attack')              # label = name of label
plt.ylabel('Defence')
plt.title('Attack Defense Scatter Plot')            # title = title of plot
```

```
Text(0.5, 1.0, 'Attack Defense Scatter Plot')
```

png

```
# Histogram
# bins = number of bar in figure
data.Speed.plot(kind = 'hist',bins = 50,figsize = (12,12))
plt.show()
```

png

```
# clf() = cleans it up again you can start a fresh
data.Speed.plot(kind = 'hist',bins = 50)
plt.clf()
# We cannot see plot due to clf()
```

```
<Figure size 432x288 with 0 Axes>
```

DICTIONARY

Why we need dictionary?

- It has 'key' and 'value'
- Faster than lists

What is key and value. Example:
- dictionary = {'spain' : 'madrid'}
- Key is spain.
- Values is madrid.

It's that easy.

Lets practice some other properties like keys(), values(), update, add, check, remove key, remove all entries and remove dicrionary.

```
#create dictionary and look its keys and values
dictionary = {'spain' : 'madrid','usa' : 'vegas'}
print(dictionary.keys())
print(dictionary.values())
```

```
dict_keys(['spain', 'usa'])
dict_values(['madrid', 'vegas'])
```

```
# Keys have to be immutable objects like string, boolean, float, integer or tubles
# List is not immutable
# Keys are unique
dictionary['spain'] = "barcelona"    # update existing entry
print(dictionary)
dictionary['france'] = "paris"       # Add new entry
print(dictionary)
del dictionary['spain']              # remove entry with key 'spain'
print(dictionary)
print('france' in dictionary)        # check include or not
dictionary.clear()                   # remove all entries in dict
print(dictionary)
```

```
{'spain': 'barcelona', 'usa': 'vegas'}
{'spain': 'barcelona', 'usa': 'vegas', 'france': 'paris'}
{'usa': 'vegas', 'france': 'paris'}
True
{}
```

```
# In order to run all code you need to take comment this line
# del dictionary         # delete entire dictionary
print(dictionary)       # it gives error because dictionary is deleted
```

```
{}
```

PANDAS

What we need to know about pandas?

- CSV: comma - separated values

```
data = pd.read_csv('pokemon.csv')
```

```
series = data['Defense']        # data['Defense'] = series
print(type(series))
data_frame = data[['Defense']]  # data[['Defense']] = data frame
print(type(data_frame))
```

```
<class 'pandas.core.series.Series'>
<class 'pandas.core.frame.DataFrame'>
```

Before continue with pandas, we need to learn **logic, control flow** and **filtering.**

Comparison operator: ==, <, >, <=

Boolean operators: and, or ,not

Filtering pandas

```
# Comparison operator
print(3 > 2)
print(3!=2)
# Boolean operators
print(True and False)
print(True or False)
```

```
True
True
False
True
```

```
# 1 - Filtering Pandas data frame
x = data['Defense']>200     # There are only 3 pokemons who have higher defense value than 200
data[x]
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	#	Name	Type 1	Type 2	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Legendary
224	225	Mega Steelix	Steel	Ground	75	125	230	55	95	30	2	False
230	231	Shuckle	Bug	Rock	20	10	230	10	230	5	2	False
333	334	Mega Aggron	Steel	NaN	70	140	230	60	80	50	3	False

```
# 2 - Filtering pandas with logical_and
# There are only 2 pokemons who have higher defence value than 2oo and higher attack value than 100
data[np.logical_and(data['Defense']>200, data['Attack']>100 )]
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	#	Name	Type 1	Type 2	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Legendary
224	225	Mega Steelix	Steel	Ground	75	125	230	55	95	30	2	False
333	334	Mega Aggron	Steel	NaN	70	140	230	60	80	50	3	False

```
# This is also same with previous code line. Therefore we can also use '&' for filtering.
data[(data['Defense']>200) & (data['Attack']>100)]
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	#	Name	Type 1	Type 2	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Legendary
224	225	Mega Steelix	Steel	Ground	75	125	230	55	95	30	2	False
333	334	Mega Aggron	Steel	NaN	70	140	230	60	80	50	3	False

WHILE and FOR LOOPS

We will learn most basic while and for loops

```
# Stay in loop if condition( i is not equal 5) is true
i = 0
while i != 5 :
    print('i is: ',i)
    i +=1
print(i,' is equal to 5')
```

```
i is:  0
i is:  1
i is:  2
i is:  3
i is:  4
5  is equal to 5
```

```
# Stay in loop if condition( i is not equal 5) is true
lis = [1,2,3,4,5]
for i in lis:
    print('i is: ',i)
print('')

# Enumerate index and value of list
# index : value = 0:1, 1:2, 2:3, 3:4, 4:5
for index, value in enumerate(lis):
    print(index," : ",value)
print('')

# For dictionaries
# We can use for loop to achive key and value of dictionary. We learnt key and value at dictionary part.
dictionary = {'spain':'madrid','france':'paris'}
for key,value in dictionary.items():
    print(key," : ",value)
print('')

# For pandas we can achieve index and value
for index,value in data[['Attack']][0:1].iterrows():
    print(index," : ",value)
```

```
i is:  1
i is:  2
i is:  3
i is:  4
i is:  5

0  :  1
1  :  2
2  :  3
3  :  4
4  :  5

spain  :  madrid
france  :  paris

0  :  Attack    49
Name: 0, dtype: int64
```

In this part, you learn:

- how to import csv file
- plotting line,scatter and histogram
- basic dictionary features
- basic pandas features like filtering that is actually something always used and main for being data scientist
- While and for loops

2. PYTHON DATA SCIENCE TOOLBOX

USER DEFINED FUNCTION

What we need to know about functions:

- docstrings: documentation for functions. Example:

  for f():

"""This is docstring for documentation of function f"""

- tuble: sequence of immutable python objects.

  cant modify values

  tuble uses paranthesis like tuble = (1,2,3)

  unpack tuble into several variables like a,b,c = tuble

```
# example of what we learn above
def tuble_ex():
    """ return defined t tuble"""
    t = (1,2,3)
    return t
a,b,c = tuble_ex()
print(a,b,c)
```

```
1 2 3
```

SCOPE

What we need to know about scope:

- global: defined main body in script
- local: defined in a function
- built in scope: names in predefined built in scope module such as print, len

  Lets make some basic examples

```
# guess print what
x = 2
def f():
    x = 3
    return x
print(x)      # x = 2 global scope
print(f())    # x = 3 local scope
```

```
2
3
```

```
# What if there is no local scope
x = 5
def f():
    y = 2*x        # there is no local scope x
    return y
print(f())         # it uses global scope x
# First local scopesearched, then global scope searched, if two of them cannot be found lastly built in scope searched.
```

```
10
```

```
# How can we learn what is built in scope
import builtins
dir(builtins)
```

```
['ArithmeticError',
 'AssertionError',
 'AttributeError',
 'BaseException',
 'BlockingIOError',
 'BrokenPipeError',
 'BufferError',
 'BytesWarning',
 'ChildProcessError',
 'ConnectionAbortedError',
 'ConnectionError',
 'ConnectionRefusedError',
 'ConnectionResetError',
```

```
 'DeprecationWarning',
 'EOFError',
 'Ellipsis',
 'EnvironmentError',
 'Exception',
 'False',
 'FileExistsError',
 'FileNotFoundError',
 'FloatingPointError',
 'FutureWarning',
 'GeneratorExit',
 'IOError',
 'ImportError',
 'ImportWarning',
 'IndentationError',
 'IndexError',
 'InterruptedError',
 'IsADirectoryError',
 'KeyError',
 'KeyboardInterrupt',
 'LookupError',
 'MemoryError',
 'ModuleNotFoundError',
 'NameError',
 'None',
 'NotADirectoryError',
 'NotImplemented',
 'NotImplementedError',
 'OSError',
 'OverflowError',
 'PendingDeprecationWarning',
 'PermissionError',
 'ProcessLookupError',
 'RecursionError',
 'ReferenceError',
 'ResourceWarning',
 'RuntimeError',
 'RuntimeWarning',
 'StopAsyncIteration',
 'StopIteration',
 'SyntaxError',
 'SyntaxWarning',
 'SystemError',
 'SystemExit',
 'TabError',
 'TimeoutError',
 'True',
 'TypeError',
 'UnboundLocalError',
 'UnicodeDecodeError',
 'UnicodeEncodeError',
 'UnicodeError',
 'UnicodeTranslateError',
 'UnicodeWarning',
 'UserWarning',
 'ValueError',
 'Warning',
 'WindowsError',
 'ZeroDivisionError',
 '__IPYTHON__',
 '__build_class__',
 '__debug__',
 '__doc__',
 '__import__',
 '__loader__',
 '__name__',
 '__package__',
 '__pybind11_internals_v3_msvc__',
 '__spec__',
 'abs',
 'all',
 'any',
 'ascii',
 'bin',
 'bool',
 'breakpoint',
 'bytearray',
 'bytes',
 'callable',
 'chr',
 'classmethod',
 'compile',
 'complex',
```

```
 'copyright',
 'credits',
 'delattr',
 'dict',
 'dir',
 'display',
 'divmod',
 'enumerate',
 'eval',
 'exec',
 'filter',
 'float',
 'format',
 'frozenset',
 'get_ipython',
 'getattr',
 'globals',
 'hasattr',
 'hash',
 'help',
 'hex',
 'id',
 'input',
 'int',
 'isinstance',
 'issubclass',
 'iter',
 'len',
 'license',
 'list',
 'locals',
 'map',
 'max',
 'memoryview',
 'min',
 'next',
 'object',
 'oct',
 'open',
 'ord',
 'pow',
 'print',
 'property',
 'range',
 'repr',
 'reversed',
 'round',
 'set',
 'setattr',
 'slice',
 'sorted',
 'staticmethod',
 'str',
 'sum',
 'super',
 'tuple',
 'type',
 'vars',
 'zip']
```

NESTED FUNCTION

- function inside function.
- There is a LEGB rule that is search local scope, enclosing function, global and built in scopes, respectively.

```
#nested function
def square():
    """ return square of value """
    def add():
        """ add two local variable """
        x = 2
        y = 3
        z = x + y
        return z
    return add()**2
print(square())
```

```
25
```

DEFAULT and FLEXIBLE ARGUMENTS

- Default argument example:

  def f(a, b=1):
      """ b = 1 is default argument"""
- Flexible argument example:

  def f(*args):
      """ *args can be one or more"""

  def f(** kwargs)
      """ **kwargs is a dictionary"""

lets write some code to practice

```
# default arguments
def f(a, b = 1, c = 2):
    y = a + b + c
    return y
print(f(5))
# what if we want to change default arguments
print(f(5,4,3))
```

```
8
12
```

```
# flexible arguments *args
def f(*args):
    for i in args:
        print(i)
f(1)
print("")
f(1,2,3,4)
# flexible arguments **kwargs that is dictionary
def f(**kwargs):
    """ print key and value of dictionary"""
    for key, value in kwargs.items():               # If you do not understand this part turn for loop part and look at dictionary in
for loop
        print(key, " ", value)
f(country = 'spain', capital = 'madrid', population = 123456)
```

```
1

1
2
3
4
country   spain
capital   madrid
population   123456
```

LAMBDA FUNCTION

Faster way of writing function

```
# lambda function
square = lambda x: x**2     # where x is name of argument
print(square(4))
tot = lambda x,y,z: x+y+z   # where x,y,z are names of arguments
print(tot(1,2,3))
```

```
16
6
```

ANONYMOUS FUNCTİON

Like lambda function but it can take more than one arguments.

- map(func,seq) : applies a function to all the items in a list

```
number_list = [1,2,3]
y = map(lambda x:x**2,number_list)
print(list(y))
```

```
[1, 4, 9]
```

ITERATORS

- iterable is an object that can return an iterator
- iterable: an object with an associated iter() method

  example: list, strings and dictionaries
- iterator: produces next value with next() method

```
# iteration example
name = "ronaldo"
it = iter(name)
print(next(it))    # print next iteration
print(*it)         # print remaining iteration
```

```
r
o n a l d o
```

zip(): zip lists

```
# zip example
list1 = [1,2,3,4]
list2 = [5,6,7,8]
z = zip(list1,list2)
print(z)
z_list = list(z)
print(z_list)
```

```
<zip object at 0x000001DC5E3D0748>
[(1, 5), (2, 6), (3, 7), (4, 8)]
```

```
un_zip = zip(*z_list)
un_list1,un_list2 = list(un_zip) # unzip returns tuble
print(un_list1)
print(un_list2)
print(type(un_list2))
```

```
(1, 2, 3, 4)
(5, 6, 7, 8)
<class 'tuple'>
```

LIST COMPREHENSİON

One of the most important topic of this kernel

We use list comprehension for data analysis often.

list comprehension: collapse for loops for building lists into a single line

Ex: num1 = [1,2,3] and we want to make it num2 = [2,3,4]. This can be done with for loop. However it is unnecessarily long. We can make it one line code that is list comprehension.

```
# Example of list comprehension
num1 = [1,2,3]
num2 = [i + 1 for i in num1 ]
print(num2)
```

```
[2, 3, 4]
```

[i + 1 for i in num1]: list of comprehension

i +1: list comprehension syntax

for i in num1: for loop syntax

i: iterator

num1: iterable object

```
# Conditionals on iterable
num1 = [5,10,15]
num2 = [i**2 if i == 10 else i-5 if i < 7 else i+5 for i in num1]
print(num2)
```

```
[0, 100, 20]
```

```
# lets return pokemon csv and make one more list comprehension example
# lets classify pokemons whether they have high or low speed. Our threshold is average speed.
threshold = sum(data.Speed)/len(data.Speed)
data["speed_level"] = ["high" if i > threshold else "low" for i in data.Speed]
data.loc[:10,["speed_level","Speed"]] # we will learn loc more detailed later
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	speed_level	Speed
0	low	45
1	low	60
2	high	80
3	high	80
4	low	65
5	high	80
6	high	100
7	high	100
8	high	100
9	low	43
10	low	58

Up to now, you learn

- User defined function
- Scope
- Nested function
- Default and flexible arguments
- Lambda function
- Anonymous function
- Iterators
- List comprehension

3.CLEANING DATA

DIAGNOSE DATA for CLEANING

We need to diagnose and clean data before exploring.

Unclean data:

- Column name inconsistency like upper-lower case letter or space between words
- missing data
- different language

We will use head, tail, columns, shape and info methods to diagnose data

```
data = pd.read_csv('pokemon.csv')
data.head()  # head shows first 5 rows
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	#	Name	Type 1	Type 2	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Legendary
0	1	Bulbasaur	Grass	Poison	45	49	49	65	65	45	1	False
1	2	Ivysaur	Grass	Poison	60	62	63	80	80	60	1	False
2	3	Venusaur	Grass	Poison	80	82	83	100	100	80	1	False
3	4	Mega Venusaur	Grass	Poison	80	100	123	122	120	80	1	False
4	5	Charmander	Fire	NaN	39	52	43	60	50	65	1	False

```
# tail shows last 5 rows
data.tail()
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	#	Name	Type 1	Type 2	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Legendary
795	796	Diancie	Rock	Fairy	50	100	150	100	150	50	6	True
796	797	Mega Diancie	Rock	Fairy	50	160	110	160	110	110	6	True
797	798	Hoopa Confined	Psychic	Ghost	80	110	60	150	130	70	6	True
798	799	Hoopa Unbound	Psychic	Dark	80	160	60	170	130	80	6	True
799	800	Volcanion	Fire	Water	80	110	120	130	90	70	6	True

```
# columns gives column names of features
data.columns
```

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

```
# shape gives number of rows and columns in a tuble
data.shape
```

```
(800, 12)
```

```
# info gives data type like dataframe, number of sample or row, number of feature or column, feature types and memory usage
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 800 entries, 0 to 799
Data columns (total 12 columns):
 #   Column      Non-Null Count  Dtype 
---  ------      --------------  ----- 
 0   #           800 non-null    int64 
 1   Name        799 non-null    object
 2   Type 1      800 non-null    object
 3   Type 2      414 non-null    object
 4   HP          800 non-null    int64 
 5   Attack      800 non-null    int64 
 6   Defense     800 non-null    int64 
 7   Sp. Atk     800 non-null    int64 
 8   Sp. Def     800 non-null    int64 
 9   Speed       800 non-null    int64 
 10  Generation  800 non-null    int64 
 11  Legendary   800 non-null    bool  
dtypes: bool(1), int64(8), object(3)
memory usage: 69.7+ KB
```

EXPLORATORY DATA ANALYSIS

value_counts(): Frequency counts

outliers: the value that is considerably higher or lower from rest of the data

- Lets say value at 75% is Q3 and value at 25% is Q1.
- Outlier are smaller than Q1 - 1.5(Q3-Q1) and bigger than Q3 + 1.5(Q3-Q1). (Q3-Q1) = IQR

  We will use describe() method. Describe method includes:
- count: number of entries
- mean: average of entries
- std: standart deviation
- min: minimum entry
- 25%: first quantile
- 50%: median or second quantile
- 75%: third quantile
- max: maximum entry

What is quantile?

- 1,4,5,6,8,9,11,12,13,14,15,16,17
- The median is the number that is in **middle** of the sequence. In this case it would be 11.
- The lower quartile is the median in between the smallest number and the median i.e. in between 1 and 11, which is 6.
- The upper quartile, you find the median between the median and the largest number i.e. between 11 and 17, which will be 14 according to the question above.

```
# For example lets look frequency of pokemom types
print(data['Type 1'].value_counts(dropna =False))  # if there are nan values that also be counted
# As it can be seen below there are 112 water pokemon or 70 grass pokemon
```

```
Water       112
Normal       98
Grass        70
Bug          69
Psychic      57
Fire         52
Rock         44
Electric     44
Dragon       32
Ghost        32
Ground       32
Dark         31
```

```
Poison       28
Fighting     27
Steel        27
Ice          24
Fairy        17
Flying        4
Name: Type 1, dtype: int64
```

```
# For example max HP is 255 or min defense is 5
data.describe() #ignore null entries
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	#	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation
count	800.0000	800.000000	800.000000	800.000000	800.000000	800.000000	800.000000	800.00000
mean	400.5000	69.258750	79.001250	73.842500	72.820000	71.902500	68.277500	3.32375
std	231.0844	25.534669	32.457366	31.183501	32.722294	27.828916	29.060474	1.66129
min	1.0000	1.000000	5.000000	5.000000	10.000000	20.000000	5.000000	1.00000
25%	200.7500	50.000000	55.000000	50.000000	49.750000	50.000000	45.000000	2.00000
50%	400.5000	65.000000	75.000000	70.000000	65.000000	70.000000	65.000000	3.00000
75%	600.2500	80.000000	100.000000	90.000000	95.000000	90.000000	90.000000	5.00000
max	800.0000	255.000000	190.000000	230.000000	194.000000	230.000000	180.000000	6.00000

VISUAL EXPLORATORY DATA ANALYSIS

- Box plots: visualize basic statistics like outliers, min/max or quantiles

```
# For example: compare attack of pokemons that are legendary  or not
# Black line at top is max
# Blue line at top is 75%
# Red line is median (50%)
# Blue line at bottom is 25%
# Black line at bottom is min
# There are no outliers
data.boxplot(column='Attack',by = 'Legendary')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x1dc5e6633c8>
```

png

TIDY DATA

We tidy data with melt().
Describing melt is confusing. Therefore lets make example to understand it.

```
# Firstly I create new data from pokemons data to explain melt nore easily.
data_new = data.head()    # I only take 5 rows into new data
data_new
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	#	Name	Type 1	Type 2	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Legendary
0	1	Bulbasaur	Grass	Poison	45	49	49	65	65	45	1	False
1	2	Ivysaur	Grass	Poison	60	62	63	80	80	60	1	False
2	3	Venusaur	Grass	Poison	80	82	83	100	100	80	1	False
3	4	Mega Venusaur	Grass	Poison	80	100	123	122	120	80	1	False
4	5	Charmander	Fire	NaN	39	52	43	60	50	65	1	False

```
# lets melt
# id_vars = what we do not wish to melt
# value_vars = what we want to melt
melted = pd.melt(frame=data_new,id_vars = 'Name', value_vars= ['Attack','Defense'])
melted
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	Name	variable	value
0	Bulbasaur	Attack	49
1	Ivysaur	Attack	62
2	Venusaur	Attack	82
3	Mega Venusaur	Attack	100
4	Charmander	Attack	52
5	Bulbasaur	Defense	49
6	Ivysaur	Defense	63
7	Venusaur	Defense	83
8	Mega Venusaur	Defense	123
9	Charmander	Defense	43

PIVOTING DATA

Reverse of melting.

```
# Index is name
# I want to make that columns are variable
# Finally values in columns are value
melted.pivot(index = 'Name', columns = 'variable',values='value')
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

variable	Attack	Defense
Name		
Bulbasaur	49	49
Charmander	52	43
Ivysaur	62	63
Mega Venusaur	100	123
Venusaur	82	83

CONCATENATING DATA

We can concatenate two dataframe

```
# Firstly lets create 2 data frame
data1 = data.head()
data2= data.tail()
conc_data_row = pd.concat([data1,data2],axis =0,ignore_index =True) # axis = 0 : adds dataframes in row
conc_data_row
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	#	Name	Type 1	Type 2	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Legendary
0	1	Bulbasaur	Grass	Poison	45	49	49	65	65	45	1	False
1	2	Ivysaur	Grass	Poison	60	62	63	80	80	60	1	False
2	3	Venusaur	Grass	Poison	80	82	83	100	100	80	1	False
3	4	Mega Venusaur	Grass	Poison	80	100	123	122	120	80	1	False
4	5	Charmander	Fire	NaN	39	52	43	60	50	65	1	False
5	796	Diancie	Rock	Fairy	50	100	150	100	150	50	6	True
6	797	Mega Diancie	Rock	Fairy	50	160	110	160	110	110	6	True
7	798	Hoopa Confined	Psychic	Ghost	80	110	60	150	130	70	6	True
8	799	Hoopa Unbound	Psychic	Dark	80	160	60	170	130	80	6	True
9	800	Volcanion	Fire	Water	80	110	120	130	90	70	6	True

```
data1 = data['Attack'].head()
data2= data['Defense'].head()
conc_data_col = pd.concat([data1,data2],axis =1) # axis = 0 : adds dataframes in row
conc_data_col
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	Attack	Defense
0	49	49
1	62	63
2	82	83
3	100	123
4	52	43

DATA TYPES

There are 5 basic data types: object(string),booleab, integer, float and categorical.

We can make conversion data types like from str to categorical or from int to float

Why is category important:

- make dataframe smaller in memory
- can be utilized for anlaysis especially for sklear(we will learn later)

```
data.dtypes
```

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

```
# lets convert object(str) to categorical and int to float.
data['Type 1'] = data['Type 1'].astype('category')
data['Speed'] = data['Speed'].astype('float')
```

```
# As you can see Type 1 is converted from object to categorical
# And Speed ,s converted from int to float
data.dtypes
```

```
#                int64
Name            object
Type 1        category
Type 2          object
HP               int64
Attack           int64
Defense          int64
Sp. Atk          int64
Sp. Def          int64
Speed          float64
Generation       int64
Legendary         bool
dtype: object
```

MISSING DATA and TESTING WITH ASSERT

If we encounter with missing data, what we can do:

- leave as is
- drop them with dropna()

- fill missing value with fillna()
- fill missing values with test statistics like mean

Assert statement: check that you can turn on or turn off when you are done with your testing of the program

```
# Lets look at does pokemon data have nan value
# As you can see there are 800 entries. However Type 2 has 414 non-null object so it has 386 null object.
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 800 entries, 0 to 799
Data columns (total 12 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   #           800 non-null    int64
 1   Name        799 non-null    object
 2   Type 1      800 non-null    category
 3   Type 2      414 non-null    object
 4   HP          800 non-null    int64
 5   Attack      800 non-null    int64
 6   Defense     800 non-null    int64
 7   Sp. Atk     800 non-null    int64
 8   Sp. Def     800 non-null    int64
 9   Speed       800 non-null    float64
 10  Generation  800 non-null    int64
 11  Legendary   800 non-null    bool
dtypes: bool(1), category(1), float64(1), int64(7), object(2)
memory usage: 65.0+ KB
```

```
# Lets chech Type 2
data["Type 2"].value_counts(dropna =False)
# As you can see, there are 386 NAN value
```

```
NaN         386
Flying       97
Ground       35
Poison       34
Psychic      33
Fighting     26
Grass        25
Fairy        23
Steel        22
Dark         20
Dragon       18
Ice          14
Ghost        14
Rock         14
Water        14
Fire         12
Electric      6
Normal        4
Bug           3
Name: Type 2, dtype: int64
```

```
# Lets drop nan values
data1=data   # also we will use data to fill missing value so I assign it to data1 variable
data1["Type 2"].dropna(inplace = True)  # inplace = True means we do not assign it to new variable. Changes automatically assigned to data
# So does it work ?
```

```
#  Lets check with assert statement
# Assert statement:
assert 1==1 # return nothing because it is true
```

```
# In order to run all code, we need to make this line comment
# assert 1==2 # return error because it is false
```

```
assert  data['Type 2'].notnull().all() # returns nothing because we drop nan values
```

```
data["Type 2"].fillna('empty',inplace = True)
```

```
assert  data['Type 2'].notnull().all() # returns nothing because we do not have nan values
```

```
# # With assert statement we can check a lot of thing. For example
# assert data.columns[1] == 'Name'
# assert data.Speed.dtypes == np.int
```

In this part, you learn:

- Diagnose data for cleaning
- Exploratory data analysis
- Visual exploratory data analysis
- Tidy data
- Pivoting data
- Concatenating data
- Data types
- Missing data and testing with assert

4. PANDAS FOUNDATION

REVİEW of PANDAS

As you notice, I do not give all idea in a same time. Although, we learn some basics of pandas, we will go deeper in pandas.

- single column = series
- NaN = not a number
- dataframe.values = numpy

BUILDING DATA FRAMES FROM SCRATCH

- We can build data frames from csv as we did earlier.
- Also we can build dataframe from dictionaries
    - zip() method: This function returns a list of tuples, where the i-th tuple contains the i-th element from each of the argument sequences or iterables.
- Adding new column
- Broadcasting: Create new column and assign a value to entire column

```
# data frames from dictionary
country = ["Spain","France"]
population = ["11","12"]
list_label = ["country","population"]
list_col = [country,population]
zipped = list(zip(list_label,list_col))
data_dict = dict(zipped)
df = pd.DataFrame(data_dict)
df
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	country	population
0	Spain	11
1	France	12

```
# Add new columns
df["capital"] = ["madrid","paris"]
df
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	country	population	capital
0	Spain	11	madrid
1	France	12	paris

```
# Broadcasting
df["income"] = 0 #Broadcasting entire column
df
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	country	population	capital	income
0	Spain	11	madrid	0
1	France	12	paris	0

VISUAL EXPLORATORY DATA ANALYSIS

- Plot
- Subplot
- Histogram:
    - bins: number of bins
    - range(tuble): min and max values of bins
    - normed(boolean): normalize or not
    - cumulative(boolean): compute cumulative distribution

```
# Plotting all data
data1 = data.loc[:,["Attack","Defense","Speed"]]
data1.plot()
# it is confusing
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x1dc5e67f508>
```

png

```
# subplots
data1.plot(subplots = True)
plt.show()
```

png

```
# scatter plot
data1.plot(kind = "scatter",x="Attack",y = "Defense")
plt.show()
```

png

```
# hist plot
data1.plot(kind = "hist",y = "Defense",bins = 50,range= (0,250),normed = True)
```

```
C:\Users\Ashish Patel\AppData\Local\Continuum\miniconda3\envs\python3.7\lib\site-packages\pandas\plotting\_matplotlib\hist.py:59:
MatplotlibDeprecationWarning:
The 'normed' kwarg was deprecated in Matplotlib 2.1 and will be removed in 3.1. Use 'density' instead.
  n, bins, patches = ax.hist(y, bins=bins, bottom=bottom, **kwds)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x1dc5e5f8e88>
```

png

```
# histogram subplot with non cumulative and cumulative
fig, axes = plt.subplots(nrows=2,ncols=1)
data1.plot(kind = "hist",y = "Defense",bins = 50,range= (0,250),normed = True,ax = axes[0])
data1.plot(kind = "hist",y = "Defense",bins = 50,range= (0,250),normed = True,ax = axes[1],cumulative = True)
plt.savefig('graph.png')
plt
```

```
<module 'matplotlib.pyplot' from 'C:\\Users\\Ashish Patel\\AppData\\Local\\Continuum\\miniconda3\\envs\\python3.7\\lib\\site-
packages\\matplotlib\\pyplot.py'>
```

png

STATISTICAL EXPLORATORY DATA ANALYSIS

I already explained it at previous parts. However lets look at one more time.

- count: number of entries
- mean: average of entries
- std: standart deviation
- min: minimum entry
- 25%: first quantile
- 50%: median or second quantile
- 75%: third quantile
- max: maximum entry

```
data.describe()
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	#	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation
count	800.0000	800.000000	800.000000	800.000000	800.000000	800.000000	800.000000	800.00000
mean	400.5000	69.258750	79.001250	73.842500	72.820000	71.902500	68.277500	3.32375
std	231.0844	25.534669	32.457366	31.183501	32.722294	27.828916	29.060474	1.66129
min	1.0000	1.000000	5.000000	5.000000	10.000000	20.000000	5.000000	1.00000
25%	200.7500	50.000000	55.000000	50.000000	49.750000	50.000000	45.000000	2.00000
50%	400.5000	65.000000	75.000000	70.000000	65.000000	70.000000	65.000000	3.00000
75%	600.2500	80.000000	100.000000	90.000000	95.000000	90.000000	90.000000	5.00000
max	800.0000	255.000000	190.000000	230.000000	194.000000	230.000000	180.000000	6.00000

INDEXING PANDAS TIME SERIES

- datetime = object
- parse_dates(boolean): Transform date to ISO 8601 (yyyy-mm-dd hh:mm:ss) format

```
time_list = ["1992-03-08","1992-04-12"]
print(type(time_list[1])) # As you can see date is string
# however we want it to be datetime object
datetime_object = pd.to_datetime(time_list)
print(type(datetime_object))
```

```
<class 'str'>
<class 'pandas.core.indexes.datetimes.DatetimeIndex'>
```

```
# close warning
import warnings
warnings.filterwarnings("ignore")
# In order to practice lets take head of pokemon data and add it a time list
data2 = data.head()
date_list = ["1992-01-10","1992-02-10","1992-03-10","1993-03-15","1993-03-16"]
datetime_object = pd.to_datetime(date_list)
data2["date"] = datetime_object
# lets make date as index
data2= data2.set_index("date")
data2
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	#	Name	Type 1	Type 2	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Legendary
date												
1992-01-10	1	Bulbasaur	Grass	Poison	45	49	49	65	65	45.0	1	False
1992-02-10	2	Ivysaur	Grass	Poison	60	62	63	80	80	60.0	1	False
1992-03-10	3	Venusaur	Grass	Poison	80	82	83	100	100	80.0	1	False
1993-03-15	4	Mega Venusaur	Grass	Poison	80	100	123	122	120	80.0	1	False
1993-03-16	5	Charmander	Fire	NaN	39	52	43	60	50	65.0	1	False

```
# Now we can select according to our date index
print(data2.loc["1993-03-16"])
print(data2.loc["1992-03-10":"1993-03-16"])
```

```
#                     5
Name         Charmander
Type 1             Fire
Type 2              NaN
HP                   39
Attack               52
Defense              43
Sp. Atk              60
Sp. Def              50
Speed                65
Generation            1
Legendary         False
Name: 1993-03-16 00:00:00, dtype: object
            #           Name Type 1  Type 2  HP  Attack  Defense  Sp. Atk  \
date
1992-03-10  3       Venusaur  Grass  Poison  80      82       83      100
1993-03-15  4  Mega Venusaur  Grass  Poison  80     100      123      122
1993-03-16  5     Charmander   Fire     NaN  39      52       43       60

            Sp. Def  Speed  Generation  Legendary
date
1992-03-10      100   80.0           1      False
1993-03-15      120   80.0           1      False
1993-03-16       50   65.0           1      False
```

RESAMPLING PANDAS TIME SERIES

- Resampling: statistical method over different time intervals
    - Needs string to specify frequency like "M" = month or "A" = year
- Downsampling: reduce date time rows to slower frequency like from daily to weekly
- Upsampling: increase date time rows to faster frequency like from daily to hourly
- Interpolate: Interpolate values according to different methods like 'linear', 'time' or index'
    - https://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.interpolate.html

```
# We will use data2 that we create at previous part
data2.resample("A").mean()
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	#	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Legendary
date									
1992-12-31	2.0	61.666667	64.333333	65.0	81.666667	81.666667	61.666667	1.0	False
1993-12-31	4.5	59.500000	76.000000	83.0	91.000000	85.000000	72.500000	1.0	False

```
# Lets resample with month
data2.resample("M").mean()
# As you can see there are a lot of nan because data2 does not include all months
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	#	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Legendary
date									
1992-01-31	1.0	45.0	49.0	49.0	65.0	65.0	45.0	1.0	0.0
1992-02-29	2.0	60.0	62.0	63.0	80.0	80.0	60.0	1.0	0.0
1992-03-31	3.0	80.0	82.0	83.0	100.0	100.0	80.0	1.0	0.0
1992-04-30	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1992-05-31	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1992-06-30	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1992-07-31	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1992-08-31	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1992-09-30	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1992-10-31	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1992-11-30	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1992-12-31	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1993-01-31	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1993-02-28	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1993-03-31	4.5	59.5	76.0	83.0	91.0	85.0	72.5	1.0	0.0

```
# In real life (data is real. Not created from us like data2) we can solve this problem with interpolate
# We can interpolete from first value
data2.resample("M").first().interpolate("linear")
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-77-d59f23fc83e7> in <module>
      1 # In real life (data is real. Not created from us like data2) we can solve this problem with interpolate
      2 # We can interpolete from first value
----> 3 data2.resample("M").first().interpolate("linear")
```

```
~\AppData\Local\Continuum\miniconda3\envs\python3.7\lib\site-packages\pandas\core\generic.py in interpolate(self, method, axis, limit, inplace, limit_direction, limit_area, downcast, **kwargs)
   7015             inplace=inplace,
   7016             downcast=downcast,
-> 7017             **kwargs,
   7018         )
   7019
```

```
~\AppData\Local\Continuum\miniconda3\envs\python3.7\lib\site-packages\pandas\core\internals\managers.py in interpolate(self, **kwargs)
    568
    569     def interpolate(self, **kwargs):
--> 570         return self.apply("interpolate", **kwargs)
    571
    572     def shift(self, **kwargs):
```

```
~\AppData\Local\Continuum\miniconda3\envs\python3.7\lib\site-packages\pandas\core\internals\managers.py in apply(self, f, filter, **kwargs)
    440                 applied = b.apply(f, **kwargs)
    441             else:
--> 442                 applied = getattr(b, f)(**kwargs)
    443             result_blocks = _extend_blocks(applied, result_blocks)
    444
```

```
~\AppData\Local\Continuum\miniconda3\envs\python3.7\lib\site-packages\pandas\core\internals\blocks.py in interpolate(self, method, axis, inplace, limit, fill_value, **kwargs)
   1887         values = self.values if inplace else self.values.copy()
   1888         return self.make_block_same_class(
-> 1889             values=values.fillna(value=fill_value, method=method, limit=limit),
   1890             placement=self.mgr_locs,
   1891         )
```

```
~\AppData\Local\Continuum\miniconda3\envs\python3.7\lib\site-packages\pandas\core\arrays\categorical.py in fillna(self, value, method, limit)
   1711         """
   1712         value, method = validate_fillna_kwargs(
-> 1713             value, method, validate_scalar_dict_value=False
   1714         )
   1715
```

```
~\AppData\Local\Continuum\miniconda3\envs\python3.7\lib\site-packages\pandas\util\_validators.py in validate_fillna_kwargs(value, method, validate_scalar_dict_value)
    332         raise ValueError("Must specify a fill 'value' or 'method'.")
    333     elif value is None and method is not None:
--> 334         method = clean_fill_method(method)
    335
    336     elif value is not None and method is None:
```

```
~\AppData\Local\Continuum\miniconda3\envs\python3.7\lib\site-packages\pandas\core\missing.py in clean_fill_method(method, allow_nearest)
     89         expecting = "pad (ffill), backfill (bfill) or nearest"
     90     if method not in valid_methods:
---> 91         raise ValueError(f"Invalid fill method. Expecting {expecting}. Got {method}")
     92     return method
     93
```

```
ValueError: Invalid fill method. Expecting pad (ffill) or backfill (bfill). Got linear
```

```
# Or we can interpolate with mean()
data2.resample("M").mean().interpolate("linear")
```

MANIPULATING DATA FRAMES WITH PANDAS

INDEXING DATA FRAMES

- Indexing using square brackets
- Using column attribute and row label
- Using loc accessor
- Selecting only some columns

```
# read data
data = pd.read_csv('pokemon.csv')
data= data.set_index("#")
data.head()
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	Name	Type 1	Type 2	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Legendary
#											
1	Bulbasaur	Grass	Poison	45	49	49	65	65	45	1	False
2	Ivysaur	Grass	Poison	60	62	63	80	80	60	1	False
3	Venusaur	Grass	Poison	80	82	83	100	100	80	1	False
4	Mega Venusaur	Grass	Poison	80	100	123	122	120	80	1	False
5	Charmander	Fire	NaN	39	52	43	60	50	65	1	False

```
# indexing using square brackets
data["HP"][1]
```

```
45
```

```
# using column attribute and row label
data.HP[1]
```

```
45
```

```
# using loc accessor
data.loc[1,["HP"]]
```

```
HP    45
Name: 1, dtype: object
```

```
# Selecting only some columns
data[["HP","Attack"]]
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	HP	Attack
#		
1	45	49
2	60	62
3	80	82
4	80	100
5	39	52
...	...	...
796	50	100
797	50	160
798	80	110
799	80	160
800	80	110

800 rows × 2 columns

SLICING DATA FRAME

- Difference between selecting columns
  - Series and data frames
- Slicing and indexing series
- Reverse slicing
- From something to end

```
# Difference between selecting columns: series and dataframes
print(type(data["HP"]))     # series
print(type(data[["HP"]]))   # data frames
```

```
<class 'pandas.core.series.Series'>
<class 'pandas.core.frame.DataFrame'>
```

```
# Slicing and indexing series
data.loc[1:10,"HP":"Defense"]   # 10 and "Defense" are inclusive
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	HP	Attack	Defense
#			
1	45	49	49
2	60	62	63
3	80	82	83
4	80	100	123
5	39	52	43
6	58	64	58
7	78	84	78
8	78	130	111
9	78	104	78
10	44	48	65

```
# Reverse slicing
data.loc[10:1:-1,"HP":"Defense"]
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	HP	Attack	Defense
#			
10	44	48	65
9	78	104	78
8	78	130	111
7	78	84	78
6	58	64	58
5	39	52	43
4	80	100	123
3	80	82	83
2	60	62	63
1	45	49	49

```
# From something to end
data.loc[1:10,"Speed":]
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	Speed	Generation	Legendary
#			
1	45	1	False
2	60	1	False
3	80	1	False
4	80	1	False
5	65	1	False
6	80	1	False
7	100	1	False
8	100	1	False
9	100	1	False
10	43	1	False

FILTERING DATA FRAMES

Creating boolean series
Combining filters
Filtering column based others

```
# Creating boolean series
boolean = data.HP > 200
data[boolean]
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	Name	Type 1	Type 2	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Legendary
#											
122	Chansey	Normal	NaN	250	5	5	35	105	50	1	False
262	Blissey	Normal	NaN	255	10	10	75	135	55	2	False

```
# Combining filters
first_filter = data.HP > 150
second_filter = data.Speed > 35
data[first_filter & second_filter]
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	Name	Type 1	Type 2	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Legendary
#											
122	Chansey	Normal	NaN	250	5	5	35	105	50	1	False
262	Blissey	Normal	NaN	255	10	10	75	135	55	2	False
352	Wailord	Water	NaN	170	90	45	90	45	60	3	False
656	Alomomola	Water	NaN	165	75	80	40	45	65	5	False

```
# Filtering column based others
data.HP[data.Speed<15]
```

```
#
231     20
360     45
487     50
496    135
659     44
Name: HP, dtype: int64
```

TRANSFORMING DATA

- Plain python functions
- Lambda function: to apply arbitrary python function to every element
- Defining column using other columns

```
# Plain python functions
def div(n):
    return n/2
data.HP.apply(div)
```

```
#
1      22.5
2      30.0
3      40.0
4      40.0
5      19.5
       ...
796    25.0
797    25.0
798    40.0
799    40.0
800    40.0
Name: HP, Length: 800, dtype: float64
```

```
# Or we can use lambda function
data.HP.apply(lambda n : n/2)
```

```
#
1      22.5
2      30.0
3      40.0
4      40.0
5      19.5
       ...
796    25.0
797    25.0
798    40.0
799    40.0
800    40.0
Name: HP, Length: 800, dtype: float64
```

```
# Defining column using other columns
data["total_power"] = data.Attack + data.Defense
data.head()
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	Name	Type 1	Type 2	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Legendary	total_power
#												
1	Bulbasaur	Grass	Poison	45	49	49	65	65	45	1	False	98
2	Ivysaur	Grass	Poison	60	62	63	80	80	60	1	False	125
3	Venusaur	Grass	Poison	80	82	83	100	100	80	1	False	165
4	Mega Venusaur	Grass	Poison	80	100	123	122	120	80	1	False	223
5	Charmander	Fire	NaN	39	52	43	60	50	65	1	False	95

INDEX OBJECTS AND LABELED DATA

index: sequence of label

```
# our index name is this:
print(data.index.name)
# lets change it
data.index.name = "index_name"
data.head()
```

```
#
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	Name	Type 1	Type 2	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Legendary	total_power
index_name												
1	Bulbasaur	Grass	Poison	45	49	49	65	65	45	1	False	98
2	Ivysaur	Grass	Poison	60	62	63	80	80	60	1	False	125
3	Venusaur	Grass	Poison	80	82	83	100	100	80	1	False	165
4	Mega Venusaur	Grass	Poison	80	100	123	122	120	80	1	False	223
5	Charmander	Fire	NaN	39	52	43	60	50	65	1	False	95

```python
# Overwrite index
# if we want to modify index we need to change all of them.
data.head()
# first copy of our data to data3 then change index
data3 = data.copy()
# lets make index start from 100. It is not remarkable change but it is just example
data3.index = range(100,900,1)
data3.head()
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	Name	Type 1	Type 2	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Legendary	total_power
100	Bulbasaur	Grass	Poison	45	49	49	65	65	45	1	False	98
101	Ivysaur	Grass	Poison	60	62	63	80	80	60	1	False	125
102	Venusaur	Grass	Poison	80	82	83	100	100	80	1	False	165
103	Mega Venusaur	Grass	Poison	80	100	123	122	120	80	1	False	223
104	Charmander	Fire	NaN	39	52	43	60	50	65	1	False	95

```python
# We can make one of the column as index. I actually did it at the beginning of manipulating data frames with pandas section
# It was like this
# data= data.set_index("#")
# also you can use
# data.index = data["#"]
```

HIERARCHICAL INDEXING

- Setting indexing

```python
# lets read data frame one more time to start from beginning
data = pd.read_csv('pokemon.csv')
data.head()
# As you can see there is index. However we want to set one or more column to be index
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	#	Name	Type 1	Type 2	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Legendary
0	1	Bulbasaur	Grass	Poison	45	49	49	65	65	45	1	False
1	2	Ivysaur	Grass	Poison	60	62	63	80	80	60	1	False
2	3	Venusaur	Grass	Poison	80	82	83	100	100	80	1	False
3	4	Mega Venusaur	Grass	Poison	80	100	123	122	120	80	1	False
4	5	Charmander	Fire	NaN	39	52	43	60	50	65	1	False

```python
# Setting index : type 1 is outer type 2 is inner index
data1 = data.set_index(["Type 1","Type 2"])
data1.head(100)
# data1.loc["Fire","Flying"] # howw to use indexes
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

		#	Name	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Legendary
Type 1	**Type 2**										
Grass	**Poison**	1	Bulbasaur	45	49	49	65	65	45	1	False
	Poison	2	Ivysaur	60	62	63	80	80	60	1	False
	Poison	3	Venusaur	80	82	83	100	100	80	1	False
	Poison	4	Mega Venusaur	80	100	123	122	120	80	1	False
Fire	**NaN**	5	Charmander	39	52	43	60	50	65	1	False
...	**...**	...	...	...	...	...	...	...	...	...	...
Poison	**NaN**	96	Grimer	80	80	50	40	50	25	1	False
	NaN	97	Muk	105	105	75	65	100	50	1	False
Water	**NaN**	98	Shellder	30	65	100	45	25	40	1	False
	Ice	99	Cloyster	50	95	180	85	45	70	1	False
Ghost	**Poison**	100	Gastly	30	35	30	100	35	80	1	False

100 rows × 10 columns

PIVOTING DATA FRAMES

- pivoting: reshape tool

```
dic = {"treatment":["A","A","B","B"],"gender":["F","M","F","M"],"response":[10,45,5,9],"age":[15,4,72,65]}
df = pd.DataFrame(dic)
df
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	treatment	gender	response	age
0	A	F	10	15
1	A	M	45	4
2	B	F	5	72
3	B	M	9	65

```
# pivoting
df.pivot(index="treatment",columns = "gender",values="response")
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

gender	F	M
treatment		
A	10	45
B	5	9

STACKING and UNSTACKING DATAFRAME

- deal with multi label indexes
- level: position of unstacked index
- swaplevel: change inner and outer level index position

```
df1 = df.set_index(["treatment","gender"])
df1
# lets unstack it
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

		response	age
treatment	**gender**		
A	**F**	10	15
	M	45	4
B	**F**	5	72
	M	9	65

```
# level determines indexes
df1.unstack(level=0)
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead tr th {
    text-align: left;
}

.dataframe thead tr:last-of-type th {
    text-align: right;
}
```

	response		age	
treatment	**A**	**B**	**A**	**B**
gender				
F	10	5	15	72
M	45	9	4	65

```
df1.unstack(level=1)
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead tr th {
    text-align: left;
}

.dataframe thead tr:last-of-type th {
    text-align: right;
}
```

	response		**age**	
gender	**F**	**M**	**F**	**M**
treatment				
A	10	45	15	4
B	5	9	72	65

```
# change inner and outer level index position
df2 = df1.swaplevel(0,1)
df2
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

		response	**age**
gender	**treatment**		
F	**A**	10	15
M	**A**	45	4
F	**B**	5	72
M	**B**	9	65

MELTING DATA FRAMES

- Reverse of pivoting

```
df
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	treatment	gender	response	age
0	A	F	10	15
1	A	M	45	4
2	B	F	5	72
3	B	M	9	65

```
# df.pivot(index="treatment",columns = "gender",values="response")
pd.melt(df,id_vars="treatment",value_vars=["age","response"])
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	treatment	variable	value
0	A	age	15
1	A	age	4
2	B	age	72
3	B	age	65
4	A	response	10
5	A	response	45
6	B	response	5
7	B	response	9

CATEGORICALS AND GROUPBY

```
# We will use df
df
```


```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```


	treatment	gender	response	age
0	A	F	10	15
1	A	M	45	4
2	B	F	5	72
3	B	M	9	65

```
# according to treatment take means of other features
df.groupby("treatment").mean()   # mean is aggregation / reduction method
# there are other methods like sum, std,max or min
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	response	age
treatment		
A	27.5	9.5
B	7.0	68.5

```
# we can only choose one of the feature
df.groupby("treatment").age.max()
```

```
treatment
A    15
B    72
Name: age, dtype: int64
```

```
# Or we can choose multiple features
df.groupby("treatment")[["age","response"]].min()
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	age	response
treatment		
A	4	10
B	65	5

```
df.info()
# as you can see gender is object
# However if we use groupby, we can convert it categorical data.
# Because categorical data uses less memory, speed up operations like groupby
#df["gender"] = df["gender"].astype("category")
#df["treatment"] = df["treatment"].astype("category")
#df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4 entries, 0 to 3
Data columns (total 4 columns):
 #   Column     Non-Null Count  Dtype 
---  ------     --------------  ----- 
 0   treatment  4 non-null      object
 1   gender     4 non-null      object
 2   response   4 non-null      int64 
 3   age        4 non-null      int64 
dtypes: int64(2), object(2)
memory usage: 256.0+ bytes
```

CONCLUSION

Thank you for your votes and comments

MACHINE LEARNING https://www.kaggle.com/kanncaa1/machine-learning-tutorial-for-beginners/

DEEP LEARNING https://www.kaggle.com/kanncaa1/deep-learning-tutorial-for-beginners

STATISTICAL LEARNING https://www.kaggle.com/kanncaa1/statistical-learning-tutorial-for-beginners

If you have any question or suggest, I will be happy to hear it.