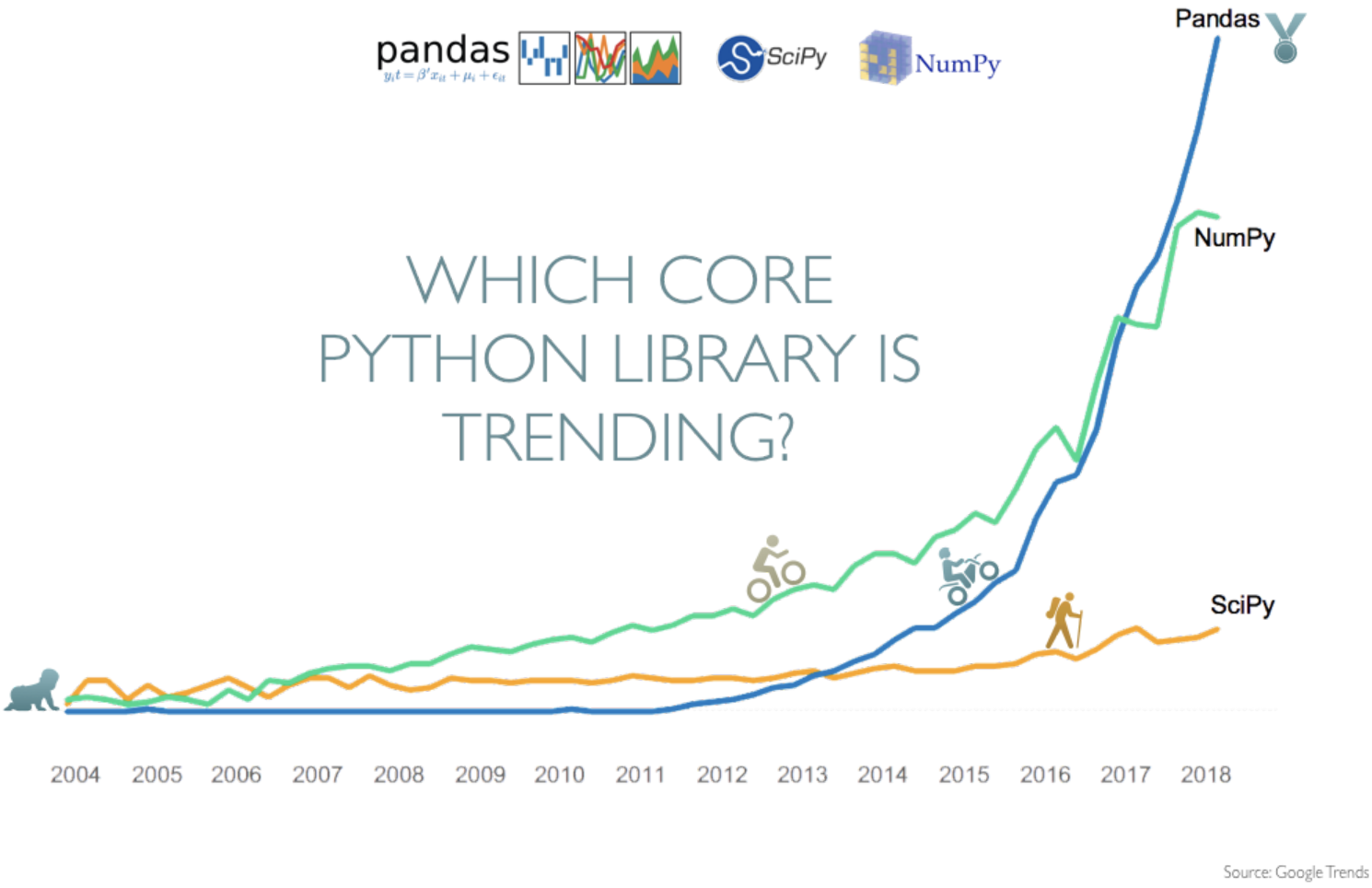
Why pandas for data analysis?

An important step for any data scientist is to prepare and clean the data so that it could be used later for deeper analysis. Analysis of raw and unclean data could lead to wrong insights which could derail the organization. Real 'raw' data needs a lot of 'wrangling' operations before it can be ready for dissection by a data scientist. One of the popular tools for data wrangling in python is `pandas`.

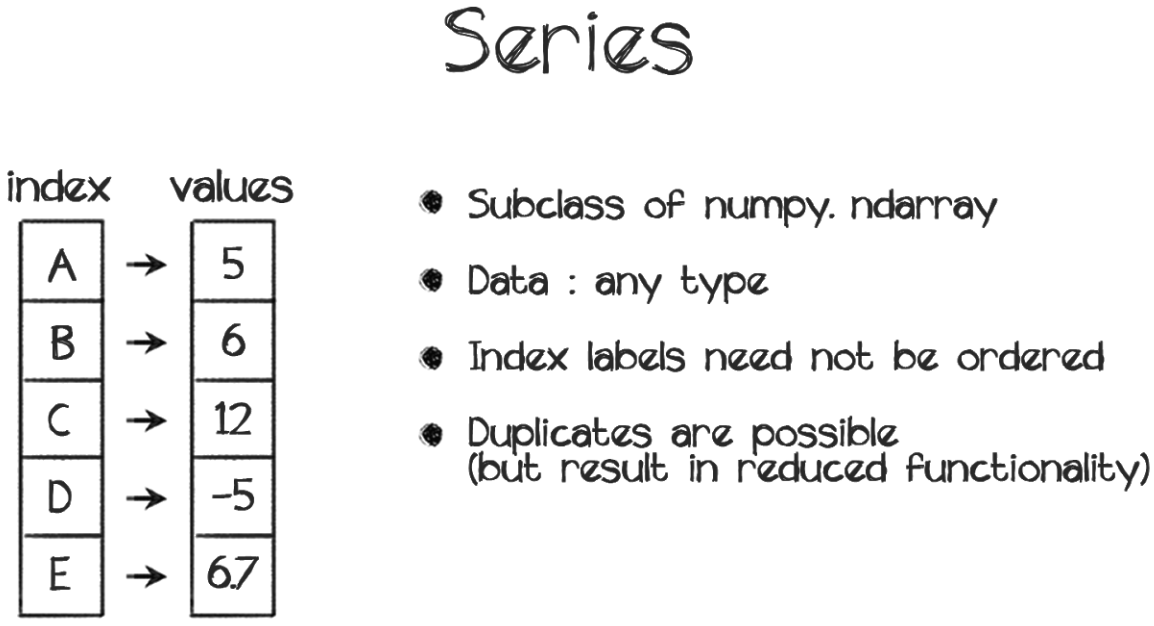
One of the reasons why Python is used extensively for Data Science is the availability of widespread packages for almost every possible function. The library `pandas` is one such package which makes life easier especially for data analysis through its extensive in-built functions for manipulations and visualizations. It is built on top of the `NumPy` and `matplotlib` library and one can harness the power of both these libraries in tandem with the power of `pandas`. Besides, there is huge community support especially for `pandas` and there are very high chances that you will get an answer for your query (if and when you get stuck) on the Internet. Take a look at the trend below to realize its popularity amongst users.



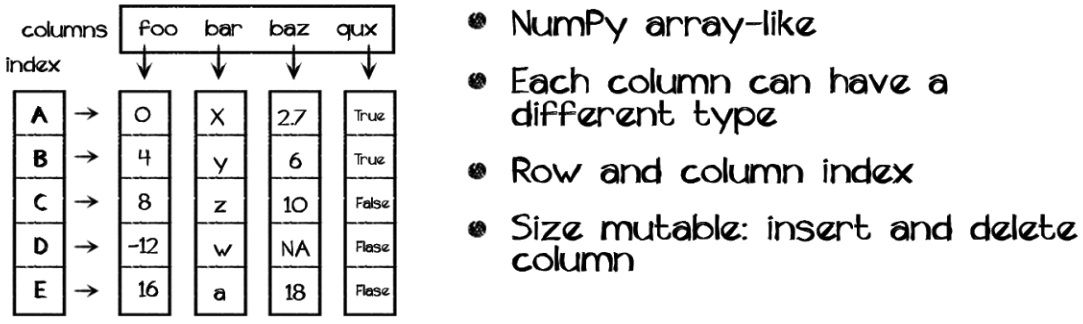
Data structures in pandas

Pandas deals with three data structures:

- Series (Labeled 1 dimensional with homogeneous/heterogeneous data but immutable size)

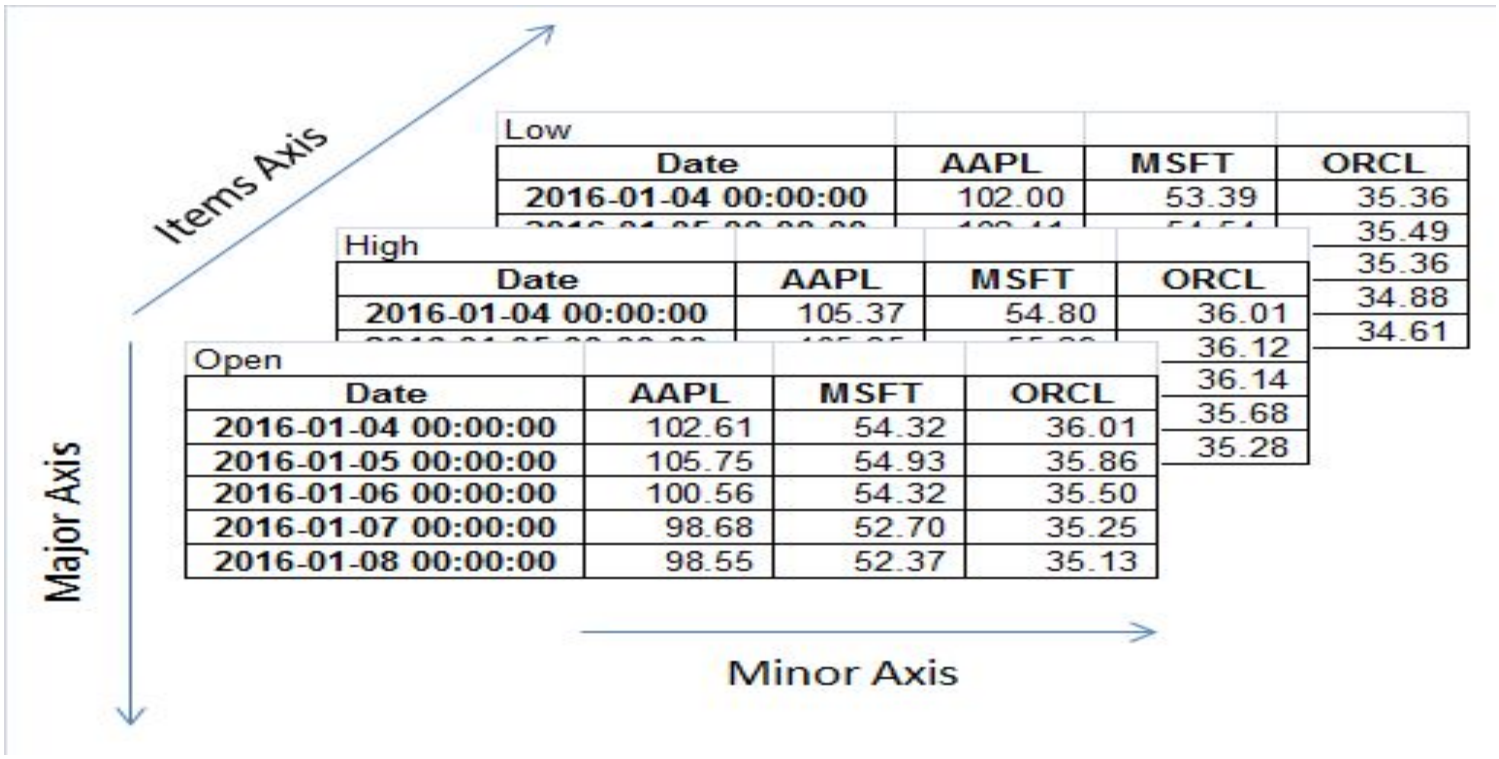


- DataFrame (Labeled 2-dimensional size-mutable tabular structure with potentially heterogeneously typed columns)



- Panel (Labeled 3 dimensional size-mutable array)

The one with that we mostly deal with in our day-to-day life is the DataFrame type which is nothing but tabular data that we frequently encounter, particularly while using Excel.



In the next part of the course, you will learn more about the Series and DataFrame variant of `pandas` data structures.

Pandas Series

A Series is a single vector of data (like a `NumPy` array) with an index that labels each element in the vector. The main difference between a series and NumPy array is that series may have axis labels, a NumPy array doesn't.

A `NumPy` array comprises the values of the series, while the index is a pandas `Index` object. It can hold any type of data (integer, string, float, python objects, etc.) as long as the data is homogeneous throughout the series. If an index is not specified, a default sequence of integers is assigned as the index.

Creating series

The constructor for series is: `pandas.Series(data, index, dtype, copy)`

Here,

- data: data (can be lists, ndarrays, dictionaries etc.)
- index: unique, hashable and same length as data (default is `np.arange(n)` where n is length of data)
- dtype: data type of series values

- copy: copy data (default `False`)

Now let's look at the different ways to create a series using `pandas`:

From NumPy ndarray

- In the first series, indices are from 0 to 2
- In the second series, they are as specified by `['a', 'b', 'c']`

```
import numpy as np
import pandas as pd

labels = ['a','b','c']
my_data = [10,20,30]
arr = np.array(my_data)

print(pd.Series(my_data))
print('==================')
print(pd.Series(my_data,index=labels))
```

Output

```
0    10
1    20
2    30
dtype: int64
==================
a    10
b    20
c    30
dtype: int64
```

If you initialize a series object with the help of `NumPy` then you can hold only homogeneous data within it.

From Dictionary

- When the index is not specified, then the keys are taken in a sorted order as index values.
- If the index is passed, values in data corresponding to the labels in the index will be accessed, the index which is absent in the keys of the dictionary will have NaN values.
- You can store heterogeneous data while creating a series with a dictionary

```
# list of lables
lables = ['a','b','c']

# dictionary
dic = {'b':1,'c':2,'d':3}

# Series without specified labels
print(pd.Series(dic))
print('============')
```

```
# Series with specified labels
print(pd.Series(dic, labels))
```

Output

```
b    1
c    2
d    3
dtype: int64
============
a    NaN
b    1.0
c    2.0
dtype: float64
```

From Scalar

- The index is provided and scalar will be repeated to match the value and the length of it.

```
# Scalar number
num = 10

# Series with index ['a','b','c']
print(pd.Series(num,index=['a','b','c']))
print('================')

# Series with index [0,1,2,3,4]
print(pd.Series(num,index=range(5)))
```

```
a    10
b    10
c    10
dtype: int64
================
0    10
1    10
2    10
3    10
4    10
dtype: int64
```

Locating data in a series is of prime importance in data analysis tasks. There are two ways by which you can access data in series objects:

By Position

A series is very similar to a `NumPy` array (index starts at 0), so data can be accessed in the same manner as we did for `NumPy` arrays. The syntax remains the same i.e. `series[start:stop:step]`. Let us understand with an example.

```
# series of numbers from 11 to 20
ser = pd.Series(data = range(11,21),index=range(10))

# retrieve the first element
print("First element is",ser[0])
print('==========')

#retrieve the first three elements
# ser[:3] -----> first three elements
print("First three elements are",ser[:3].values)
print('==========')

# retrieve index
print(ser.index)
print('==========')

# retrieve data
print(ser.values)
print('==========')
```

Output

```
First element is 11
==========
First three elements are [11 12 13]
==========
RangeIndex(start=0, stop=10, step=1)
==========
[11 12 13 14 15 16 17 18 19 20]
==========
```

By labels

We can also use the index labels to access data given the condition that the label is in the index; otherwise, it will throw a `KeyError`. Accessing data can be either:

- Single element access: `series[index]`
- Multiple element access: `series[[index1, index2, index3, .....]]`

Remember to use [[ ]] to access multiple elements via labels

The example below shows accessing data with labels

```
# series of first five multiples of 10
ser = pd.Series(data = [10,20,30,40,50], index = ['a','b','c','d','e'])

# retrieve value at index 'b'
print("Value at index 'b' is ",ser['b'])
print('==========')

# retrieve value at indexes 'a','c' and 'e'
print("Values at indexes 'a','c' and 'e' are ", ser[['a','c','e']].values)
print('==========')

#retrieve value at index 'f' (not present)
```

```
try:
    print("Value at index 'f' is",ser['f'])
except KeyError:
    print("There is no such index")
```

Output

```
Value at index 'b' is  20
==========
Values at indexes 'a','c' and 'e' are  [10 30 50]
==========
There is no such index
```

NOTE: It may happen that while combining series you arrive at another series which contains null values or NaNs. Most often they are undesirable and you can replace them with any value you want with the help of `.fillna()` method of pandas. Let's say you want to replace NaNs with value `a`; simply use `series.fillna(a)` and additionally, if you want the change to be permanent use `inplace=True` inside `.fillna()` method.

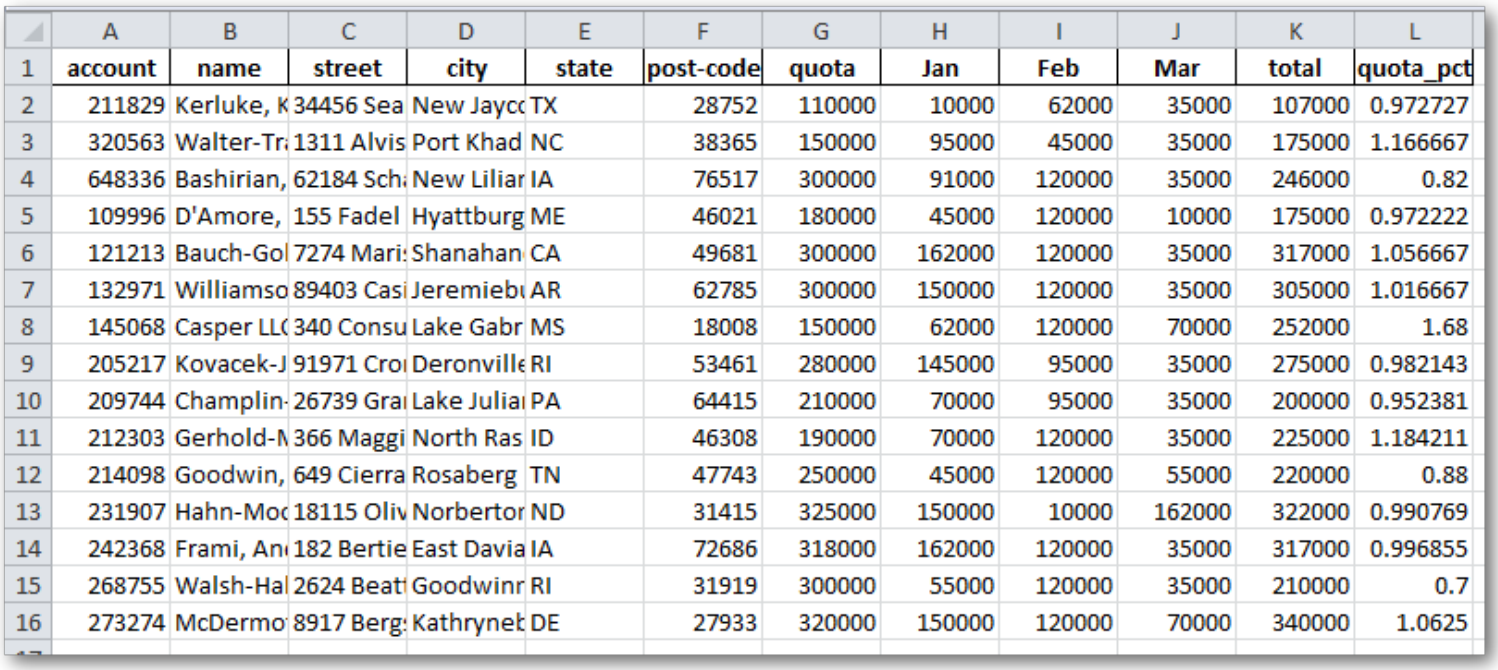
Example:
In this task, you will add two series, with different indices.

- Create two series with same values [1,2,3,4] but having different indices; one with [0,1,2,3] and the other [0,1,3,4]. Save the first series as `a` and the second one as `b`
- Then add both the series as `a+b` and save it as `c`
- Print it out to check `c` and you will observe some missing value as NaNs
- Replace that NaNs permanently with `0`s using `.fillna(0, inplace=True)`
- Again print out `c` and to confirm that NaNs have been replaced

```
import pandas as pd
a = pd.Series(data = [1, 2, 3, 4], index = [0,1, 2 ,3])
b = pd.Series(data = [1,2,3,4], index = [0,1,3,4])
c = a+b
print(a)
print(b)
print(c)
c.fillna(0,inplace = True)
print(c)
```

What is a dataframe?

The concept of a dataframe comes from the world of statistical software used in empirical research. It generally refers to tabular data: a data structure representing instances(rows), each of which consists of a number of measurements(columns). Alternatively, each row may be treated as a single observation of multiple variables. An example of dataframe that we commonly come across in Excel is shown below:



	A	B	C	D	E	F	G	H	I	J	K	L
1	account	name	street	city	state	post-code	quota	Jan	Feb	Mar	total	quota_pct
2	211829	Kerluke, K	34456 Sea	New Jayc	TX	28752	110000	10000	62000	35000	107000	0.972727
3	320563	Walter-Tr	1311 Alvis	Port Khad	NC	38365	150000	95000	45000	35000	175000	1.166667
4	648336	Bashirian,	62184 Sch	New Liliar	IA	76517	300000	91000	120000	35000	246000	0.82
5	109996	D'Amore,	155 Fadel	Hyattburg	ME	46021	180000	45000	120000	10000	175000	0.972222
6	121213	Bauch-Gol	7274 Mari	Shanahan	CA	49681	300000	162000	120000	35000	317000	1.056667
7	132971	Williamso	89403 Cas	Jeremiebu	AR	62785	300000	150000	120000	35000	305000	1.016667
8	145068	Casper LL	340 Consu	Lake Gabr	MS	18008	150000	62000	120000	70000	252000	1.68
9	205217	Kovacek-J	91971 Cro	Deronville	RI	53461	280000	145000	95000	35000	275000	0.982143
10	209744	Champlin-	26739 Gra	Lake Julia	PA	64415	210000	70000	95000	35000	200000	0.952381
11	212303	Gerhold-M	366 Maggi	North Ras	ID	46308	190000	70000	120000	35000	225000	1.184211
12	214098	Goodwin,	649 Cierra	Rosaberg	TN	47743	250000	45000	120000	55000	220000	0.88
13	231907	Hahn-Moc	18115 Oliv	Norbertor	ND	31415	325000	150000	10000	162000	322000	0.990769
14	242368	Frami, An	182 Bertie	East Davia	IA	72686	318000	162000	120000	35000	317000	0.996855
15	268755	Walsh-Hal	2624 Beat	Goodwinr	RI	31919	300000	55000	120000	35000	210000	0.7
16	273274	McDermo	8917 Berg	Kathrynet	DE	27933	320000	150000	120000	70000	340000	1.0625

Here,

-2,3,4,.....are the rows/instances

- `account`, `name`, `street`, `city` etc. are the measurements/variables for each instance

Features of `pandas` dataframe

Due to the widespread use of 2-D tabular data, `pandas` is one of the most widely used packages, especially for dataframes. Dataframes have the following features:

- Columns can be of different types
- Size is mutable
- Labelled axes (rows and columns)
- Can perform arithmetic operations on rows and columns

How to create dataframes?

The constructor for `pandas` dataframe object is `pandas.DataFrame( data, index, columns, dtype, copy).`

Here,

- `data`: various forms (ndarray, series, map, lists, dict, constants, another DataFrame)
- `index`: index labels (default `np.arange(n)`)
- `columns`: column names (default `np.arange(n)`); True only when `index` is not specified
- `dtype`: Data type of each column
- `copy`: copying of data (default `False`)

Now depending on the form of the `data`, we can construct dataframes from different sources. Let us discuss a few of them:

From lists

```
#import packages
import pandas as pd
import numpy as np

# list of values (single column)
data = ['Rob','Bobby','John','Danny','Manny']

#construct dataframe with column called 'Name'
df = pd.DataFrame(data, columns = ['Name'])

#display
df
```

Output

```
    Name
0    Rob
1    Bobby
2    John
3    Danny
4    Manny
```

Here, we pass the names ['Rob', 'Bobby', 'John', 'Danny', 'Manny'] as the values of a column/feature titled `Name` and having indices [0, 1, 2, 3, 4].Now let us pass a list of lists as values so that we can accomodate more than single column/feature. It is demonstrated below:

```
#list of values (two columns)
data =[['Rob',25],['Bobby',30],['John',21],['Danny',32],['Manny',23]]

#construct dataframe with columns called 'Name' and 'Age'
df = pd.DataFrame(data,columns = ['Name','Age'])

#display
df
```

Output

```
    Name    Age
0    Rob     25
1    Bobby   30
2    John    21
3    Danny   32
4    Manny   23
```

We have simply added `Age` column for each and every instance (rows)

From dictionary

We can also use dictionaries for creating dataframes. Let's see how:

- Dictionary of ndarrays/lists: The keys of the dictionary will be the feature names and the values will be the values for that feature across the dataframe. Remember that the ndarrays/lists must have the same length.

```
#data source
data = {'Name':['Rob','Bobby','John','Danny','Manny'], 'Age':[25,30,21,32,23]}

#construct dataframe
df = pd.DataFrame(data, index = ['R','B','J','D','M'])

#display
df
```

Output

```
    Age Name
R    25  Rob
B    30  Bobby
J    21  John
D    32  Danny
M    23  Manny
```

In the above example, we have constructed the same dataframe as in the previous example but using a dictionary this time, albeit with an index. Observe closely to make out the syntax.

From list of dictionaries

Here, each element corresponds to a row/instance and every element is a dictionary. This dictionary in turn contains the feature names as the keys and feature values as the values of that key. We create the same dataframe as the previous example this time but now as a list of dictionaries.

```
# data source
data = [{'Name':'Rob','Age':25},{'Name':'Bobby','Age':30},
        {'Name':'John','Age':21},{'Name':'Danny','Age':32},
        {'Name':'Manny','Age':23}]

#construct dataframe
df = pd.DataFrame(data, index=['R','B','J','D','M'])

#display
df
```

Output

```
    Age Name
R    25  Rob
B    30  Bobby
J    21  John
D    32  Danny
M    23  Manny
```

From series:

```
#construct the dataframe
df =
pd.DataFrame({'Name':pd.Series(['Rob','Bobby','John','Danny','Manny'],index=['R','B','J','D','M'])
,
                      'Age':pd.Series([25,30,21,32,23],index=['R','B','J','D','M'])})
#display
df
```

Output

```
   Age  Name
R    25   Rob
B    30   Bobby
J    21   John
D    32   Danny
M    23   Manny
```

Example:

In this task you will create a dataframe containing the phone numbers and pincodes of two persons `Richie` and `Mark` using any the methods explained in the topic.

- Save their pincodes as a list `pincodes` which has values `[800678, 800456]`
- Save their phone numbers as a list `numbers` containing values `[2567890, 2567657]`
- Create a list `labels` which you will be used as an index for the dataframe. It contains the initials of `Richie` and `Mark` i.e. `R` and `M`
- Create a dataframe with index as `labels` and columns as `Number` and `Pincode`. Save the dataframe as `first`
- Print out `first`

```
import pandas as pd
pincodes = [800678, 800456]
numbers = [2567890, 2567657]
labels = ['R','M']
names = ['Richie','Mark']
data = {'Name': names,'Number': numbers,'Pincode': pincodes}
first = pd.DataFrame(data   , index = labels)
print(first)
```

File I/O

`pandas` I/O API provides a set of reader functions like `read_csv()`, `read_table()` and returns a `pandas` object. It parses the data and converts it intelligently into a DataFrame.

If the file has a .csv format use

`pandas.read_csv(filepath_or_buffer, sep=',', delimiter=None, header='infer',names=None, index_col=None, usecols=None`

** to convert a file into a DataFrame. Most of the files you encounter will be in CSV format.

Here,

- `filepath_or_buffer`: the path to the file
- `sep`: Delimiter to use
- `delimiter`: Alternative argument name for sep (default `None`)
- `header`: Row number(s) to use as the column names, and the start of the data
- `names`: List of column names to use
- `index_col`: Column to use as the row labels of the DataFrame
- `usecols`: Return a subset of the columns

Quick Exploration of Data

You will be working on the 'pokemon.csv' data for your tasks and we will provide a snapshot of operations which were carried out on a subset of the data.

About Pokemons: The Millenials must be well acquainted with Pokémons. For those of you that do not know about them, we will provide you with a brief background of what Pokémons are (those who know already can skip).

Pokémon—short for pocket monsters—is the name of an anime series involving creatures called pokémon and trainers. Within the narrative of the series, the pokémon trainer catches pokémon in little holding containers (called pokeballs) and then uses that pokémon to fight other pokémon. On its surface, the fights have two reasons:

- to weaken and capture wild pokémon
- to defeat other pokémon trainers.

The pokémon themselves are various, having different appearances, names, powers, potentials, weaknesses, and personalities.

Dataset description

This data set includes 721 Pokemon, including their number, name, first and the second type, and basic stats: `HP`, `Attack`, `Defense`, `Special Attack`, `Special Defense`, and `Speed`.

Feature description:

- `#`: ID for each pokemon
- `Name`: Name of each pokemon
- `Type 1`: Each pokemon has a type, this determines weakness/resistance to attacks
- `Type 2`: Some pokemon are dual type and have 2
- `Total`: the sum of all stats that come after this, a general guide to how strong a pokemon is
- `HP`: hit points, or health, defines how much damage a Pokemon can withstand before fainting

- `Attack`: the base modifier for normal attacks (eg. Scratch, Punch)
- `Defense`: the base damage resistance against normal attacks
- `SP Atk`: special attack, the base modifier for special attacks (e.g. fire blast, bubble beam)
- `SP Def`: the base damage resistance against special attacks
- `Speed`: determines which pokemon attacks first each round

Now let us look and understand some of the functions that you will be using to have a quick glance and understanding of data.

1) Looking at the top few rows: Use `.head(n)` to display first n rows. By default it displays first 5 rows.

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

Output

	#	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

2) Looking at the last few rows: Use `.tail(n)` to display the last n rows. By default, it displays the last 5 rows

```
df.tail(5)
```

Output

	#	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

3) General information of every column: Use `.info()` method to display datatypes for each column, number of non-missing values and memory usage by the dataframe.

```
df.info()
```

Output

```
  <class 'pandas.core.frame.DataFrame'>
    RangeIndex: 800 entries, 0 to 799
```

```
    Data columns (total 12 columns):
    #             800 non-null int64
    Name          799 non-null object
    Type 1        800 non-null object
    Type 2        414 non-null object
    HP            800 non-null int64
    Attack        800 non-null int64
    Defense       800 non-null int64
    Sp. Atk       800 non-null int64
    Sp. Def       800 non-null int64
    Speed         800 non-null int64
    Generation    800 non-null int64
    Legendary     800 non-null bool
    dtypes: bool(1), int64(8), object(3)
    memory usage: 69.6+ KB
```

4) Data type of every column: Use `.dtypes` attribute

```
df.dtypes
```

Output

```
   #              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
```

5) Display column names: Use `.columns` attribute to check all column names

```
df.columns
```

6) Check dimensions: To check dimensions of dataframe, use `.shape` attribute

```
df.shape
```

7) Check missing values per column: Use `.isnull().sum()` to check missing values per column

```
df.isnull().sum()
```

8) Check number of unique values per column: Use `.nunique()` to check unique values for every column

```
df.nunique()
```

9) Dropping missing values: Use `.dropna()` to drop rows with missing values from the dataframe. You can use `inplace=True` if you want to modify the dataframe in-place.

```
df.dropna()
```

Example:

In this task, you will do a quick review of the data at hand i.e `df`.

- Look at the first 10 instances using `.head(10)` on the dataframe `df` and save it as `head`. Print `head`
- Use the `.describe()` method and save it as `describe`. Print it out
- Check its dimensions with `.shape` attribute and save it to a variable `shape`. Print it out to check the shape of the dataframe
- Look at the number of missing values per attribute with `.isnull().sum()` and save it to a variable `null`. Print this one out too.
- Find the number of unique values per attribute with the `.nunique()` method and save it to a variable `unique`. As before, print `unique` out

```
# Code starts here
# head of the dataframe
head = df.head(10)
print(head)
# describe the dataframe
describe = df.describe()
print(describe)
# shape of the dataframe
shape = df.shape
print(shape)
# check for null values
null = df.isnull().sum()
print(null)
# check for unique values
unique = df.nunique()
print(unique)
# code ends here
```

Selection, Creation, and Deletion

Before exploring further into the data, you need to learn how to create, select and delete values according to rows and columns. Let us look at some example to understand how it works. All of the examples have been performed on the Pokemon dataset only.

Column operations

- Selection: If you have a DataFrame `df` and you want to select a column `col1` you can do it by `df[col1]`; if you have multiple columns `col1`, `col2`, `col3` you do it by `df[[col1, col2, col3]]`

```
# select column 'Name' from dataframe
df['Name']
```

Here, the column `Name` is selected and if you care to check its type, it is a Series object.

```
# select columns 'Name', 'HP' and 'Attack'
df[['Name','HP','Attack']]
```

Here, the columns `Name`, `HP` and `Attack` are selected

- Creation: Now, you want to make a new column `Difference` which is the difference between `Attack` and `Defense` column for every Pokemon. How to do it? Its actually quite simple. As you already know every column is basically a pandas series object which is again a NumPy series. Provided the data types of the series match, you can add the values by a + operator. In this case also, you will do the same and the syntax is: `df[new_column] = df[col1] + df[col2]` (You can also perform subtraction, division etc.)

```
# create column 'Difference' using 'Attack' and 'Defence'
df['Difference']=df['Attack']-df['Defense']
```

- Deletion: Now you want to delete the column `Difference` that you had just made. You can do it by `df.drop([col1, col2, ...], inplace=True, axis=0/1)`. Note that inplace=True deletes columns from the dataframe permanently and axis specifies whether to drop across columns (axis=1) or rows (axis=0)

```
# delete column 'Difference' permanently
df.drop(['Difference'],inplace=True,axis=1)
```

Row operations

- Selection: You can access rows by either label of the index using `loc` or integer (row number) using `iloc` keyword.
  Syntax using `loc`: `df.loc[index]`
  Syntax using `iloc`: `df.iloc[row number]`

Example:

```
df.iloc[0]
df.loc[0]
```

In the code above both of them point to the same row; the first row can be accessed via `iloc[0]` and via `loc[0]` since it has an index of 299. Their output is also the same and it is:

```
#                 300
Name           Taillow
Type 1          Normal
Type 2          Flying
HP                  40
Attack              55
Defense             30
Sp. Atk             30
```

```
Sp. Def             30
Speed               85
Generation           3
Legendary        False
Name: 299, dtype: object
```

- Slicing: Use `df[start:end]` to slice rows according to row number (not label); here end value is not inclusive. Heres how you can slice from row numbers 2 and 3: `df[2:4]`
- Creation/Addition: Use `df.append(data)` where `data` is a DataFrame or Series/dictionary-like object, or list of these. In our Pokemon dataset, you want to add another Pokemon whose `#` value is
- 800
- 800and rest of its attributes are all 0. Lets look at how you can add this new instance:

```
df = df.append(pd.DataFrame([[800, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]], columns=df.columns))

df
```

108	109	Voltorb	Electric	NaN	40	30	50	55	55	100	1	0
97	98	Shellder	Water	NaN	30	65	100	45	25	40	1	0
142	143	Lapras	Water	Ice	130	85	80	85	95	60	1	0
213	214	Murkrow	Dark	Flying	60	85	42	85	42	91	2	0
498	499	Mega Lucario	Fighting	Steel	70	145	88	140	70	112	4	0
396	397	Glalie	Ice	NaN	80	80	80	80	80	80	3	0
459	460	Wormadam Sandy Cloak	Bug	Ground	60	79	105	59	85	36	4	0
57	58	Meowth	Normal	NaN	40	45	35	40	40	90	1	0
79	80	Tentacruel	Water	Poison	80	70	65	80	120	100	1	0
670	671	Chandelure	Ghost	Fire	60	55	90	145	90	80	5	0
602	603	Leavanny	Bug	Grass	75	103	80	70	80	92	5	0
0	800	0	0	0	0	0	0	0	0	0	0	0

81 rows × 12 columns

Observe the last row. This is the one that we had created.

- Deletion: You can delete rows using the `.drop()` to drop rows by specifying axis=0 inside the function. Also, you have the liberty to drop either by label or by position. In the example of the addition of rows, observe that the new instance has an index label of 0. Let's delete it permanently.

```
df.drop(index=0,axis=0,inplace=True)
```

Now lets check whether this row was actually deleted; we already know that its index was 0, so we will check its presence in the list of indices which are available in `df.index`

```
0 in df.index
```

Output

```
False
```

Cleaning the Data

The columns `Sp. Atk`, `Sp. Def` seem like really odd names to work with. Also, the `#` attribute doesn't seem to convey any type of information other than the fact that it is unique for every pokemon. So, the `Name` attribute is enough for describing and also it is convenient to call pokemon by their names rather than their ids.

Also, instead of row labels as `[0,1,2,3,4,...]` don't you think it would be helpful if you have Pokemons' names instead. Well, you should definitely do this!

So, in this topic we will perform three operations which are described below:

- Renaming columns: To rename columns from `col1`, `col2` to `newcol1`, `newcol2`, use the function `.rename(columns={col1:newcol1, col2:newcol2}, inplace=True)` to permanently rename the columns.
- Dropping columns: You have already learnt how to do this.
- Set index: To set index labels for column `column`, use `set_index(column, inplace=True)`
- Reset index in dataframes

Another operation which although is not covered in any of the tasks here but you would be frequently used while dealing with data is the `.reset_index()` method. In the image below in the first code snippet where we set the index of the dataframe according to the values in `'#'` column.

```
# Set '#' as index
df_2 = df.set_index('#')
df_2.head()
```

Output

#	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

Now imagine you need to revert it to the original index form due to some reasons. You can do it with the `reset_index()` method. This method will simply push the index values into a column and set default values as an index.

This method is useful when the index needs to be treated as a column, or when the index is meaningless and needs to be reset to the default before another operation.

```
# reset the index to the original form
df_2.reset_index(inplace = True)
df_2.head()
```

	#	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

Rename, drop and clean

In this task, you're going to apply some basic cleaning operations that you will encounter frequently while dealing with `pandas`. You're going to rename columns, drop columns permanently as well as re-index the dataframe.

- Rename the column `HP`, `Sp. Atk`, `Sp. Def` as `Health Points`, `Attack speed points` and `Defense speed points` respectively
- Drop the column `#` permanently from the dataframe with `.drop()` method. Put `inplace=True` and `axis=1` inside the method
- Set index to the `Name` column with `.set_index()` method. Here also, put `inplace=True` inside the method
- Now look at the first five rows using `.head()`

```
# Code starts here
# Rename columns 'HP', 'Sp. Atk' and 'Sp. Def' as 'Health Points', 'Attack speed points' and 'Defense speed points'
df.rename(columns={'HP':'Health Points','Sp. Atk':'Attack speed points','Sp. Def':'Defense speed points'},inplace=True)
# Remove the '#' column permanently
df.drop(['#'],inplace=True,axis=1)
# Set index as names
df.set_index('Name',inplace=True)
# Look at the first 5 observations
df.head()
# Code ends here
```

Exploring Categorical Data

Let's understand some functions which will help us analyse categorical columns better.

1) `.value_counts()`: It gives a quick count of observations for each level. This doesn't count NAs and can be applied on series objects; not dataframes.

2) `.unique()`: All the unique values present in the series, very similar to the `set()` function

3) `nunique()`: Length of the list returned by `.unique()` method. It is the total number of unique elements in the series

Now, we will take the help of the above three functions to answer these questions-

- How many different variants of Type 1 pokemon are there?
- What are the different variants of Type 1?
- What is the count for each variant of Type 1?

Look at the image below for the answers

```
# How many different variants of Type 1 are there
type_1 = df['Type 1'].nunique()
print(type_1)

# Different variants of Type 1 pokemon
print(df['Type 1'].unique())

# Counts for different variants of Type 1 pokemons
print(df['Type 1'].value_counts()
```

Output

```
18
['Grass' 'Fire' 'Water' 'Bug' 'Normal' 'Poison' 'Electric' 'Ground'
 'Fairy' 'Fighting' 'Psychic' 'Rock' 'Ghost' 'Ice' 'Dragon' 'Dark' 'Steel'
 'Flying']
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
```

Exploring Numerical Columns

Now you will explore the numerical attributes `'Health Points', 'Attack', 'Defense', 'Attack speed points', 'Defense speed points', 'Speed', 'Generation'`. Although they are numbers, we need to check if some of them actually represent categories. For example:- We can bin 1000 bats into 5 categories and name them as 1, 2, 3, 4 and 5. But that doesn't take away the fact that they are nothing but a category.

Let us check first which of these numeric attributes are actually categorical in nature. A simple strategy could be finding out the total number of unique values of the feature and dividing by the total number of instances of it. We will use `df[col].nunique` to calculate the number of unique values for `col` attribute. The ratio is around 0.0246 which is very low. So, it can be treated as representing a category.

```
# Check if 'Legendary' column is categorical in nature

# calculate number of unique values
len_legendary = df['Legendary'].nunique()

# divide it by length
print(len_legendary/len(df['Legendary']))
```

Output

```
0.024691358024691357
```

So, can you answer which Pokemon has the highest `Attack` value? You can use the the `df[col].idxmax()` on a feature `col` to find this out. The output of this function gives the index for which the value/values of that column is maximum. Remember in our dataframe, the names of the Pokemon are in the index.

A sample example is given below where we found the Pokemon with the highest `Attack` points:

```
# Which pokemon has the highest Attack?
max_attack = df.Attack.idxmax()
print(max_attack)
```

Output*

```
Mega Mewtwo X
```

Find the pokemon with the highest points

In this task, you will inspect which Pokemon has the highest points in health, special attack, special defense and speed.

- You can find the index for a particular column at which the maximum value occurs by `dataframe[column].idxmax()`
- Save the Pokemon with the highest speed as `fastest_pokemon`, with highest health points as `healthiest_pokemon`, highest special attack points as `special_attack_pokemon` and highest special defense points as `special_defense_pokemon`

```
# Code starts here


# Which pokemon has the highest 'Health Points'?
healthiest_pokemon = df['Health Points'].idxmax()
print(healthiest_pokemon)
# Which pokemon has the highest Special Attack points?
special_attack_pokemon = df['Attack speed points'].idxmax()
print(special_attack_pokemon)
# Which pokemon has the highest Special Defense points?
special_defense_pokemon = df['Defense speed points'].idxmax()
print(special_defense_pokemon)
```

```
# Which pokemon has highest Speed?
fastest_pokemon = df['Speed'].idxmax()
print(fastest_pokemon)
# Code ends here
```

Conditional Filtering

This is a very important way to filter out information according to some constraints. For example, We want to know in a sample of 50 people how many of them are man but are not wearing black? You will often come across such type of conditional filtering operations where you will have to filter across multiple conditions on multiple features to prepare new data.

You have already come across such a concept in NumPy where you use something like this `array[array > 5]` to mask or filter out values (in this example greater than 5). It works in the same way in `pandas` dataframes.

Let's look at an example of how it is done. Suppose you want only the Pokemons of the first generation, You can use `df[df['Generation'] == 1]` to do it which will generate a subset of data where pokemon are of first-generation.

In this task you will be answering the following questions:

- **Which type (`Type 1`) has the highest number of Legendary Pokemons?**
- **Find how many Pokemons which have a single type (`Type 1` present and `Type 2` absent)**

For the first question:

- First, use conditional filtering to get a dataframe that has only Legendary Pokemons
- On this dataframe count the total number of unique instances of `Type 1` column with the help of `dataframe['Type 1'].value_counts()`. It will give you which types have how many legendary pokemon
- To get the index with the maximum value use `.idxmax()` on the above series
- Store the result(the type with the highest legendary pokemon) in a variable called `highest_legendary`

For the second question:

- You'll create two Boolean conditions here; one will check if it has no `Type 2` and the other checks if it is `Legendary`
- For the first Boolean condition (no `Type 2`) the condition will be `dataframe['Type 2'].isnull()` which returns a Boolean series consisting of `False` values for rows having null values on `Type 2` column
- In the second condition, you can check for legendary status by `dataframe['Legendary'] == True` which returns `True` for all Legendary pokemon
- Both these conditions need to be satisfied and hence join them by `and` keyword
- Store the result
- Then apply conditional filtering on the dataframe and take its length using `len()` to get those Pokemons which have only `Type 1` and `Legendary`
- Store the result(the number of legendary pokemon who don't have type 2) in a variable called `'single_type_legendary'`

```
# Drop row with Name as nan
df = df[df.index.notnull()]
# Code starts here
# Find out which type of pokemons (use only `Type 1`) have the highest chances of
being Legendary
highest_legendary = df[df['Legendary'] == True]['Type 1'].value_counts().idxmax()
print(highest_legendary)
# Pokemons which do not have 'Type 2' but are Legendary
single_type_legendary = len(df[df['Type 2'].isnull() & df['Legendary'] == True])
print(single_type_legendary)
# Code ends here
```

Apply Functions

Functions can be applied along the axes of a DataFrame using the `.apply()` the method, which, like the descriptive statistics methods, takes an optional `axis` argument. By default, the operation performs column-wise, taking each column as an array-like.

Let us look at an example where we subtracted each value of the `Attack` column by its mean over the feature and dividing by its range (

$$\text{maximum - minimum}$$

$$maximum - minimum)$$

```
# minumum value
lower = np.min(df['Attack'])

# maximum value
upper = np.max(df['Attack'])

# range
limit = upper - lower

# mean
mean = np.mean(df['Attack'])

# function
def standardize(x,x_mean,x_range):
    return (x-x_mean)/x_range

# apply for 'Total' column
print(df['Attack'].apply(lambda x:standardize(x,mean,limit)))
```

Output (first 20 observations)

```
0     -0.162169
1     -0.091899
2      0.016209
3      0.113507
4     -0.145953
```

```
5     -0.081088
6      0.027020
7      0.275669
8      0.135128
9     -0.167574
10    -0.086493
11     0.021615
12     0.129723
13    -0.264872
14    -0.318926
15    -0.183791
16    -0.237845
17    -0.291899
18     0.059453
19     0.383777
20    -0.183791
```

In this task, you will make use of the `.apply()` function to modify the Pokemon names and both its types

- Convert index containing Pokemon names to upper case letters with `dataframe.index = dataframe.index.str.upper()`. Remember that `.apply()` cannot be used with index labels
- Convert values in `Type 1` into lowercase letters with the `.lower()` method using `apply()`
- Convert values in `Type 2` into lowercase if present and replace it with `None` if absent. The lambda function is `lambda x: x.lower() if isinstance(x, str) else None`; what it does essentially is convert the `Type 2` value to lowercase if it is a string or else it sets the value as `None`

```
# Code starts here
# Convert 'Name' to uppercase
df.index = df.index.str.upper()
# Convert 'Type 1' to lowercase
df['Type 1'] = df['Type 1'].apply(lambda x: x.lower())
# Convert 'Type 2' to lowercase if present else
df['Type 2'] = df['Type 2'].apply(lambda x: x.lower() if isinstance(x, str) else None)
```

Groupby and Sorting

Now, we want to compare `Attack speed points` across the categories of `Generation` for Pokemons. You might be tempted to use conditional filtering which in turn uses Boolean indexing to filter out values. But it has a downside; it will only result in binary outcomes i.e. either Yes or No. Writing separate functions for different categories is not advisable and here the `groupby` functions become important.

As the name suggests, the groupby function divides our dataset into groups based on our choice of the attribute. It is helpful in the sense that we can:

- Compute summary statistics for every group
- Perform group-specific transformations
- Do the filtration of data

Creating groups

In `pandas` we do it with the help of `.groupby()` function which returns a GroupBy object. Lets understand it through an example where we will group Pokemons according to `Generation`:

```
# Group by 'Generation'
df.groupby('Generation')
```

Output

```
<pandas.core.groupby.groupby.DataFrameGroupBy object at 0x1070466d8>
```

You can also decide to group based on multiple attributes, for example:

```
# Group by 'Generation' and 'Type 1'
df.groupby(['Generation','Type 1'])
```

Output

```
<pandas.core.groupby.groupby.DataFrameGroupBy object at 0x1070466a0>
```

In this example, instances will are grouped according to their `Generation` and then for every category of `Generation` are grouped by `Type 1` categories. We will see that in the later steps.

Inspecting groups

Now that we have created the groups, how to inspect them? Well, just use the `.groups` the attribute of the groupby object. It returns a dictionary where keys are the categories and values are the row labels for that category. For example, if we do `df.groupby('Generation').groups` we will get a dictionary with categories of `Generation` as keys and row labels (names of Pokemons) as values.

```
df.groupby('Generation').groups
```

Output (truncated)

```
{1: Index([       'Bulbasaur',          'Ivysaur',         'Venusaur',
           'Mega Venusaur',       'Charmander',       'Charmeleon',
               'Charizard', 'Mega Charizard X', 'Mega Charizard Y',
                'Squirtle',
        ...
                'Articuno',           'Zapdos',          'Moltres',
                 'Dratini',        'Dragonair',        'Dragonite',
                  'Mewtwo',    'Mega Mewtwo X',    'Mega Mewtwo Y',
                     'Mew'],
       dtype='object', name='Name', length=166),
 2: Index(['Chikorita', 'Bayleef', 'Meganium', 'Cyndaquil', 'Quilava',
        'Typhlosion', 'Totodile', 'Croconaw', 'Feraligatr', 'Sentret',
        ...
        'Raikou', 'Entei', 'Suicune', 'Larvitar', 'Pupitar', 'Tyranitar',
        'Mega Tyranitar', 'Lugia', 'Ho-oh', 'Celebi'],
       dtype='object', name='Name', length=106),
 3: Index(['Treecko', 'Grovyle', 'Sceptile', 'Mega Sceptile', 'Torchic',
        'Combusken', 'Blaziken', 'Mega Blaziken', 'Mudkip', 'Marshtomp',
        ...
```

Using aggregate functions on groups

The next logical step after grouping them is the operation we need to perform on these groups. Let's say we want to calculate the median value of `Sp. Atk` for every `Generation`.

```
# median value of Attack' across categories of 'Generation'
df.groupby('Generation')[['Sp. Atk']].median()
```

Output

```
Generation Sp. Atk
     1         65
     2         65
     3         70
     4         71
     5         65
     6         65
```

Another way of doing the same operation is :

```
# median value of 'Sp. Atk' across categories of 'Generation'
df.groupby('Generation').agg({'Sp. Atk':'median'})
```

Output

```
Generation Sp. Atk
     1         65
     2         65
     3         70
     4         71
     5         65
     6         65
```

Here we use `.agg()` inside which is a dictionary and the keys are the attributes and values are the statistic we want to calculate.

Sorting

Okay, now we want to sort the median value of `Sp. Atk` for every `Generation` in descending order. We can do it with the help of `.sort_values(by=column, ascending=False)` where the column is the name of the column we want to sort by and ascending=`True` if we want to sort in ascending order. Let's see how it works out in our case

```
# sort median value of 'Sp. Atk' across categories of 'Generation'
df.groupby('Generation')[['Sp. Atk']].median().sort_values(by='Sp. Atk',ascending=False)
```

Output

```
Generation Sp. Atk
     4         71
     3         70
```

```
1        65
2        65
5        65
6        65
```

In this task, you will use groupby to find the fastest Pokemons with the help of `Type 1` and `Speed` attributes. Also, you won't be using mean to aggregate speed values for a group; you will use median.

- Groupby Pokemons on `Type 1` based on the attribute `Speed` and aggregate the types based on the median values of `Speed` using `.median()`
- Then sort using `.sort_values(ascending=False)` to sort values in descending order
- Pick out the index list using `.index` which gives the types (`Type 1`) and pick up the first index. Save it as `fastest_type` and print it out

```
import numpy as np
# Code starts here
# Determine which type (Type 1) pokemons are the fastest(Speed)
fastest_type = df.groupby('Type 1')[['Speed']].median().sort_values(by='Speed',
ascending=False).index[0]
print(fastest_type)
# Code ends here
```

Creating pivot tables

You might be familiar with the concept of pivot tables (if you use Excel) which enabled users to automatically sort, count, total, or average the data stored in one table. A typical example is shown below where we add the `Attack` values `Generation`-wise

	A	B
1	Generation	Sum - Attack
2	1	12722
3	2	7635
4	3	13060
5	4	10027
6	5	13541
7	6	6216
8	**Total Result**	**63201**

The same operation can also be performed via `pandas.pivot(data, columns, index, aggfunc)` where

- `data`: dataframe to be used for pivot operation
- `columns`: Keys to group by on pivot table column

- `index`: column/array to groupby our data (Will be displayed in the index column (or columns, if you're passing in a list)
- `values` (optional): Column to aggregate (If we do not specify this then the function will aggregate all numeric columns)
- `aggfunc`: Functions to be applied to for every group (by default computes mean)

```
# pivot using pandas
pd.pivot_table(df,index='Generation',values='Attack',aggfunc='sum')
```

Output

```
Generation Attack
    1       12722
    2        7635
    3       13060
    4       10027
    5       13541
    6        6216
```

The difference in the values is due to the cleaning we did, so do not get perplexed.

Creating multi-index pivot tables

Let us observe an example where this time we will create a multi-index pivot table where we first group based on `Legendary` and then on `Generation` to find out the mean `Attack`. In other words, it gives us the mean `Attack` for every `Generation` and `Legendary` status

```
# multi-index pivot table
pd.pivot_table(df,index=['Legendary','Generation'],values='Attack')
```

Output

```
Legendary     Generation  Attack
    False     1           74.981250
              2           70.693069
              3           76.971831
              4           79.620370
              5           78.246667
              6           70.459459
    True      1          120.833333
              2           99.000000
              3          118.333333
              4          109.846154
              5          120.266667
              6          125.250000
```

Another way to achieve the same result is by using the `column` argument in the `.pivot_table()` method. If we pass `columns=Generation`, then it will group by `Legendary` and then a group for every `Generation` and then calculate mean `Attack` across all of them.

```
# multi-index pivot table
pd.pivot_table(df,index='Legendary',values='Attack',columns='Generation')
```

Output

Generation	1	2	3	4	5	6
Legendary						
False	74.981250	70.693069	76.971831	79.620370	78.246667	70.459459
True	120.833333	99.000000	118.333333	109.846154	120.266667	125.250

Merging DataFrame

What do we mean by merging of dataframes?

If you have ever worked with databases you must be familiar with the concept of merge and join operations. The same behaviour in pandas can be achieved with the help of `pandas.merge()` and also the `pandas.join()` methods. Merging two or more datasets is the process of bringing them together into one, and aligning the rows from each based on common attributes or columns.

Avoid using for loops

Using `for` loops you can also achieve the same outcome, but it is not advisable as it results in more verbose code which runs slowly as compared to the `.merge()` and `.join()` functionalities provided by `pandas` module. So, if you ever come across such a situation, avoid using loops and instead use the functionalities provided.

Syntax of merge

The syntax is `pd.merge(left, right, how='inner', on=None, left_on=None, right_on=None, left_index=False, right_index=False, sort=True)` where:

- `left`: dataframe
- `right`: dataframe
- `on`: Columns (names) to join on. Must be found in both the left and right DataFrame objects.
- `left_on`: Columns from the `left` DataFrame to use as keys (can either be column names or arrays with length equal to the length of the DataFrame).
- `right_on`: Columns from the `right` DataFrame to use as keys (can either be column names or arrays with length equal to the length of the DataFrame).
- `left_index`: If True, use the index (row labels) from the `left` DataFrame as its join key(s). In case of a DataFrame with a MultiIndex (hierarchical), the number of levels must match the number of join keys from the `right` DataFrame.
- `right_index`: Same usage as `left_index` for the `right` DataFrame.
- `how` – One of 'left', 'right', 'outer', 'inner'. Defaults to 'inner'
- `sort` – Sort the result DataFrame by the join keys in lexicographical order. Defaults to `True`.

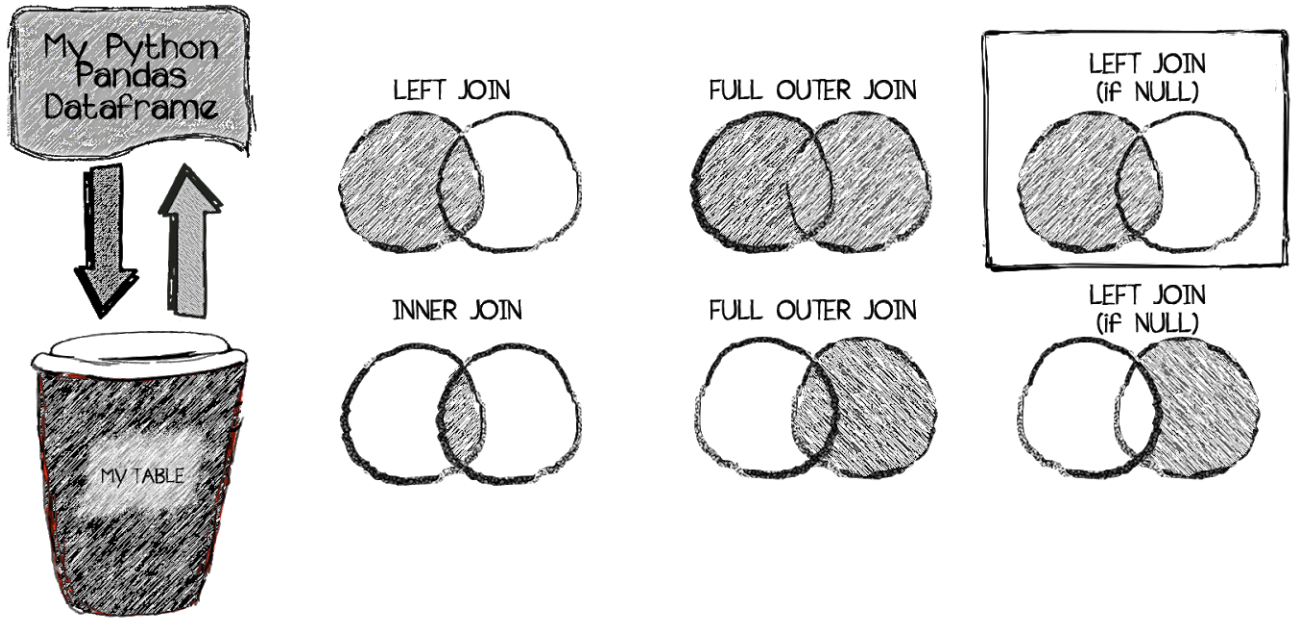
Remember that while using merge you should always specify the columns were to merge on (specified by the `on` argument inside `.merge()`)

Joins in dataframes

By default `pandas` merge function joins dataframes by inner join. An inner merge, (or inner join) keeps only the common values in both the left and right dataframes for the result. However, there are other types of joins or merge also possible with `pandas` module. Let's discuss them:

1. Inner Merge / Inner join: The default `pandas` behaviour; only keep rows where the merge on value exists in both the left and right dataframes.
2. Left Merge / Left outer join(aka left merge or left join): Keep every row in the `left` dataframe; where there are missing values of the on variable in the `right` dataframe, add empty / NaN values in the result.
3. Right Merge / Right outer join(aka right merge or right join): Keep every row in the `right` dataframe; where there are missing values of the on variable in the `left` column, add empty / NaN values in the result.
4. Outer Merge / Full outer join – A full outer join returns all the rows from the `left` dataframe, all the rows from the `right` dataframe, and matches up rows where possible, with NaNs elsewhere.

The Venn diagram below will give you a more clear understanding of the merge operations



Merge in practice

Now, let's take a look at how you can implement merging with `pandas`. Let's say we have two dataframes `attack` and `defense` describing the attacking and defensive powers of Pokemons along with their names present in both of the dataframes. The `attack` dataframe consists only of the row indices `[1,3,5,7,9]` whereas `defense` consists of the first 5 rows.

```
attack.head()
```

Output

	Name	Attack	Sp. Atk
1	Ivysaur	62	80
3	Mega Venusaur	100	122
5	Charmeleon	64	80
7	Mega Charizard X	130	130
9	Squirtle	48	50

```
defense.head()
```

Output

	Name	Defense	Sp. Def
0	Bulbasaur	49	65
1	Ivysaur	63	80
2	Venusaur	83	100
3	Mega Venusaur	123	120
4	Charmander	43	50

Now let's do some merge operations:

1. Inner merge: Both the dataframes have only two rows in common; one with Pokemon `Ivysaur` and the other `Mega Venusaur` and since inner merge picks up only those rows common to both the dataframes, so we have the following result: Only two Pokemons Ivysaur and Mega Venasaur are common to the dataframes so while doing an inner merge it contains these two rows only.

```
# Inner join
pd.merge(left=attack,right=defense,on='Name',how='inner')
```

Output

	Name	Attack	Sp. Atk	Defense	Sp. Def
0	Ivysaur	62	80	63	80
1	Mega Venusaur	100	122	123	120

2. Outer merge: It will return all the rows for both dataframes and will assign NaN values wherever the values are not present. The `attack` dataframe contains Pokemons Charmeleon, Mega Charizard X and Squirtle which don't have any presence in the `defense` dataframe and so their defense attributes are NaNs. Similarly, the instances which are in `defense` but not in `attack` will have their attack attributes represented by NaNs.

```
# Outer join
```

```
pd.merge(left=attack,right=defense,on='Name',how='outer')
```

Output

	Name	Attack	Sp. Atk	Defense	Sp. Def
0	Ivysaur	62.0	80.0	63.0	80.0
1	Mega Venusaur	100.0	122.0	123.0	120.0
2	Charmeleon	64.0	80.0	NaN	NaN
3	Mega Charizard X	130.0	130.0	NaN	NaN
4	Squirtle	48.0	50.0	NaN	NaN
5	Bulbasaur	NaN	NaN	49.0	65.0
6	Venusaur	NaN	NaN	83.0	100.0
7	Charmander	NaN	NaN	43.0	50.0

3. Left merge: It will return a dataframe with all the possible columns from both `attack` and `defense` dataframe but containing only `attack` dataframe instances. So, there is NaNs where attribute values are absent.

```
# Left join
pd.merge(left=attack,right=defense,on='Name',how='left')
```

	Name	Attack	Sp. Atk	Defense	Sp. Def
0	Ivysaur	62	80	63.0	80.0
1	Mega Venusaur	100	122	123.0	120.0
2	Charmeleon	64	80	NaN	NaN
3	Mega Charizard X	130	130	NaN	NaN
4	Squirtle	48	50	NaN	NaN

4. Right merge: It will return a dataframe with all the possible columns from both `attack` and `defense` dataframe but only for `defense` dataframe instances. So, there is NaNs where attribute values are absent

```
# Right join
pd.merge(left=attack,right=defense,on='Name',how='right')
```

Output

	Name	Attack	Sp. Atk	Defense	Sp. Def
0	Ivysaur	62.0	80.0	63	80
1	Mega Venusaur	100.0	122.0	123	120
2	Bulbasaur	NaN	NaN	49	65
3	Venusaur	NaN	NaN	83	100
4	Charmander	NaN	NaN	43	50

Can we track merges?

In the process of merging dataframes you can often get confused where a particular instance came from; thanks to `pandas` you can now keep a track of the merge. Just pass the `indicator=True` inside the `.merge()` to view this. For example:

Output

	Name	Attack	Sp. Atk	Defense	Sp. Def	_merge
0	Ivysaur	62.0	80.0	63.0	80.0	both
1	Mega Venusaur	100.0	122.0	123.0	120.0	both
2	Charmeleon	64.0	80.0	NaN	NaN	left_only
3	Mega Charizard X	130.0	130.0	NaN	NaN	left_only
4	Squirtle	48.0	50.0	NaN	NaN	left_only
5	Bulbasaur	NaN	NaN	49.0	65.0	right_only
6	Venusaur	NaN	NaN	83.0	100.0	right_only
7	Charmander	NaN	NaN	43.0	50.0	right_only

In this task you will merge two dataframes `df1` and `df2` into one single dataframe. `df1` has column `fruit` and `df2` has column `product` which contain the same entity fruits; so you will be merging both of them. In a similar manner, the columns `weight` and `kilo` on `df1` and `df2` respectively represent the weights of the fruits and so you will be merging on them as well. You will be doing an `inner` merge operation that contains rows that are common to both these dataframes on these two pairs of columns.

- From the left dataframe i.e. `df1` you will be taking two columns for merging `fruit` and `weight`; so pass them as `left_on=['fruit','weight']` inside the `.merge()` method
- Now from the right dataframe `df2` you are taking two columns for merging with the two columns from `df1`; so pass them as `right_on=['product', 'kilo']` inside `.merge()` method
- Since it is an inner join pass `how='inner'` also
- To distinguish the `price` attribute from the left and right dataframes for an instance, pass `suffixes=['_left', '_right']`

```
df1 = pd.DataFrame({'fruit': ['apple', 'banana', 'orange'] * 3,

                    'weight': ['high', 'medium', 'low'] * 3,

                    'price': np.random.randint(0, 15, 9)})
```

```
df2 = pd.DataFrame({'product': ['apple', 'orange', 'pine'] * 2,

                    'kilo': ['high', 'low'] * 3,

                    'price': np.random.randint(0, 15, 6)})

merged =
pd.merge(df1,df2,how='inner',on=None,left_on=['fruit','weight'],right_on=['product',
'kilo'],suffixes=['_left', '_right'],sort=True)

print(merged)
```

Important Takeaways from Pandas

Before we close, let us reconcile the picture we began with - converting `raw` data into a cleaner form that is conducive for further processing. During this long journey of pandas, you have done all sorts of data wrangling tasks (detecting missing values, renaming columns, dropping and replacing missing values) along with generating insights from the data. The raw data that we had was ambiguous which needed some amount of cleansing before we could gather any kind of information from it.

Before

	#	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

After

	Type 1	Type 2	Health Points	Attack	Defense	Attack speed points	Defense speed points	Speed	Generation	Legendary	Total
BULBASAUR	grass	poison	45	49	49	65	65	45	1	False	318
IVYSAUR	grass	poison	60	62	63	80	80	60	1	False	405
VENUSAUR	grass	poison	80	82	83	100	100	80	1	False	525
MEGA VENUSAUR	grass	poison	80	100	123	122	120	80	1	False	625
CHARMANDER	fire	None	39	52	43	60	50	65	1	False	309

Look at this picture and reflect on the journey till now. If you are looking for insights and understanding from your data, think of which form of data you would prefer.

- The new dataframe after all the tasks you have done has no ambiguity with the column names - the column names are more meaningful.
- A new column `Total` was also created which is the sum of all the powers for a Pokemon - a valuable addition to the previous dataframe.
- `Type 2` null values were replaced with the `None` keyword and the row with a null value for the name of the Pokemon was deleted.
- Every instance has an index as the name of the Pokemon which makes it more meaningful than random indices.

Now all of these operations were possible due to the vast functions made available by `pandas`. This dataset is now fit for doing any kind of Machine Learning task. For ex: Given other features predict whether the Pokemon is Legendary or not and many more. Sometimes data-wrangling alone might be insufficient and we might need to get visual summaries to get a better understanding of data. In the next chapter, we will look closely on how to visualize data effectively for insights.