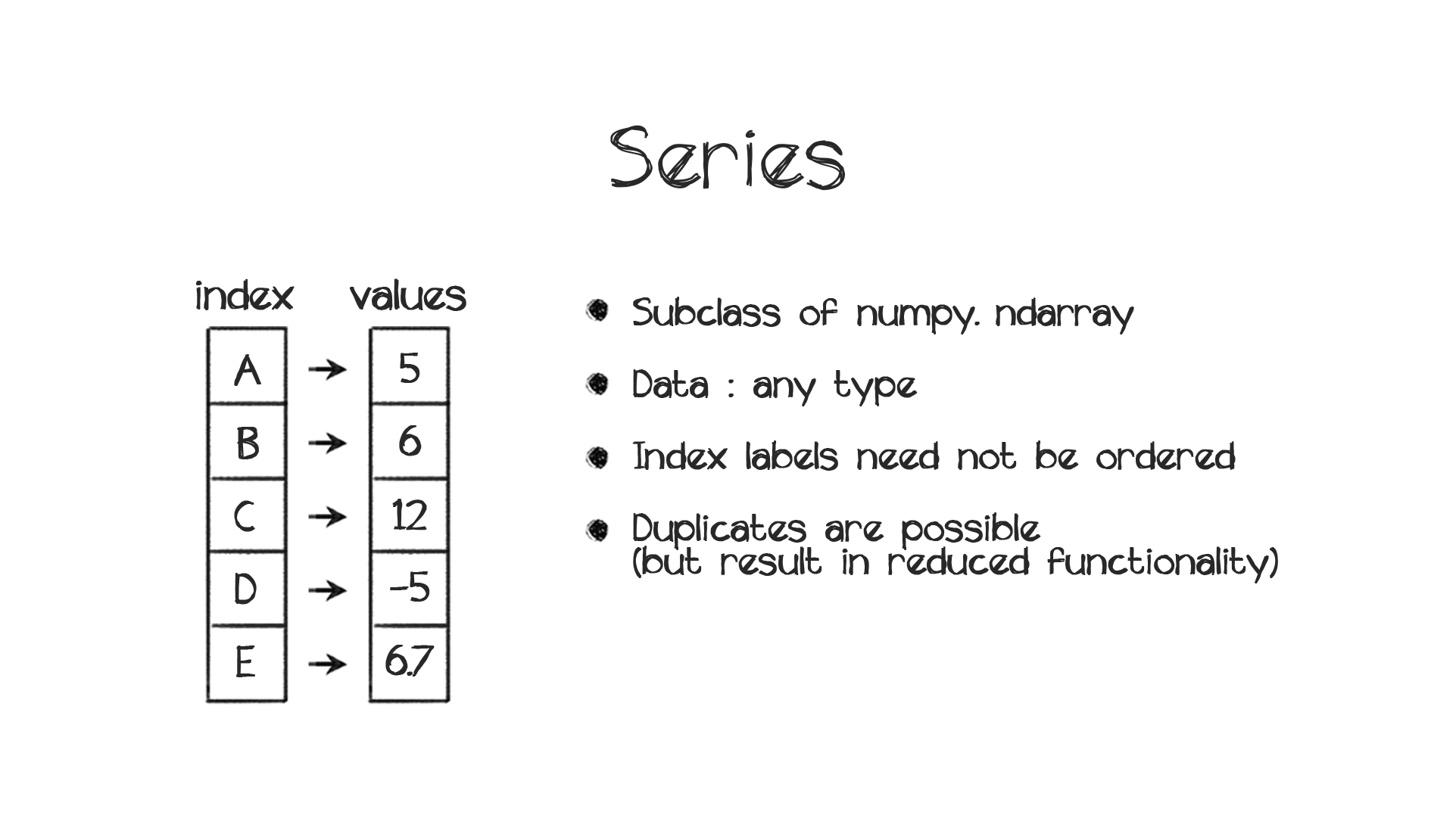
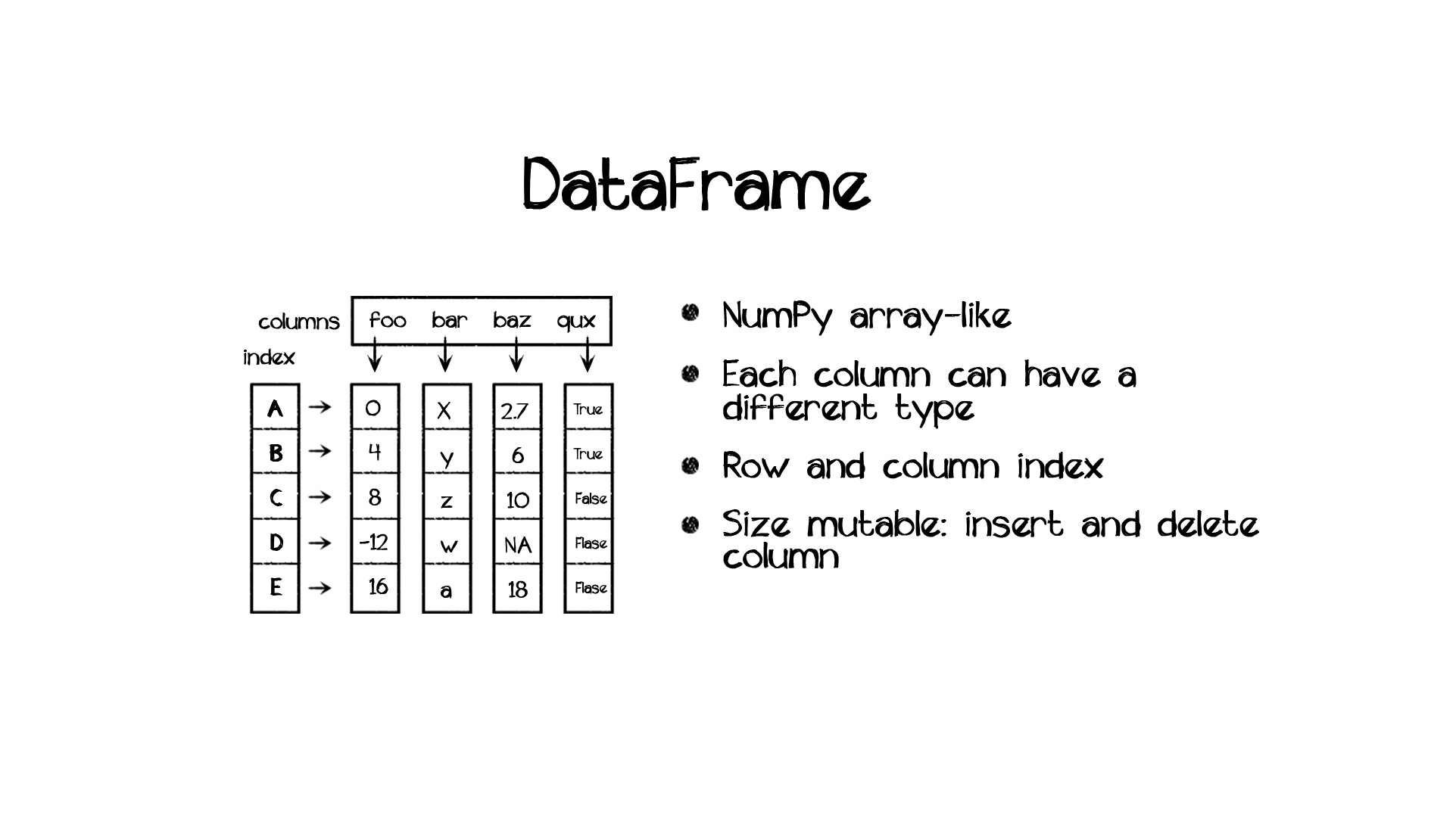


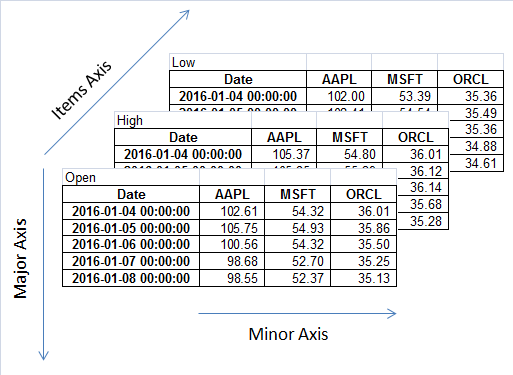
**Data structures in pandas**

Pandas deals with three data structures:

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



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**

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

Accessing Data in a Series(3/6)

50

LESSON

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]

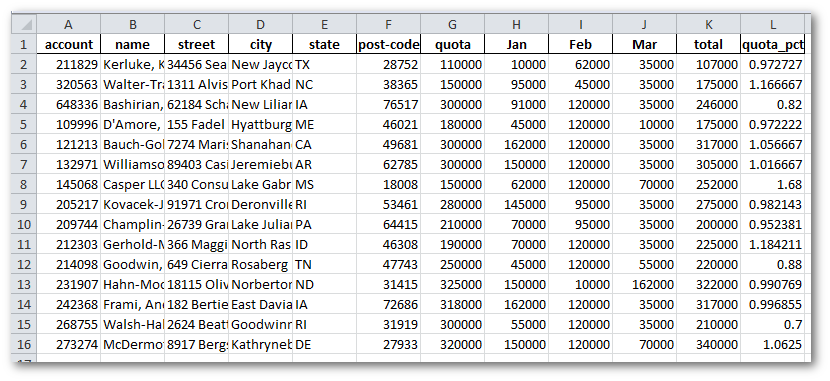
==========

There **is** no such index

**NOTE:** It may happen that while combining series you arrive at another series which contains null values or **NaN**s. 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 **NaN**s with value a; simply use series.fillna(a) and additionally, if you want the change to be permanent use inplace=True inside .fillna() method.

# ****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:



Here,

-2, 3, 4, …..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

**Basic Pandas Operations**

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(2/7)

50

LESSON

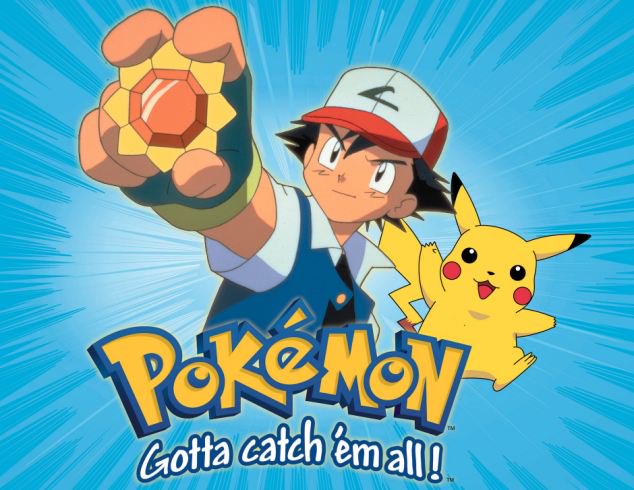
You will be working on the **'pokemon.csv'** data for your tasks and we will provide a snapshot of operations which were carried out on a subset of the data.

**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**:

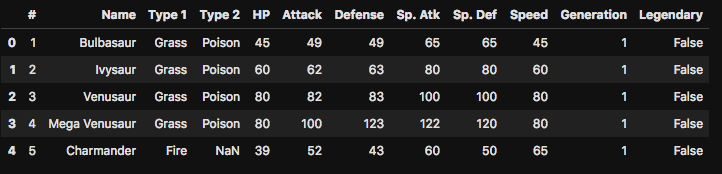
* #: ID for each pokemon
* Name: Name of each pokemon
* Type 1: Each pokemon has a type, this determines weakness/resistance to attacks
* Type 2: Some pokemon are dual type and have 2
* Total: the sum of all stats that come after this, a general guide to how strong a pokemon is
* HP: hit points, or health, 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)

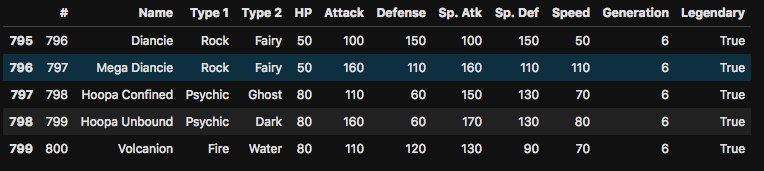
**Output**



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**



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()

Selection, Creation, and Deletion(3/7)

50

LESSON

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

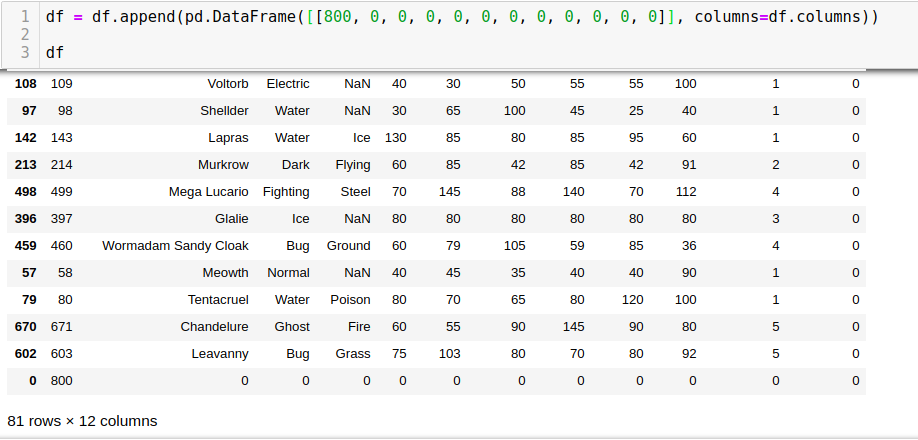
Speed 85

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 is800800and rest of its attributes are all 0. Lets look at how you can add this new instance:



**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**

Cool! This instance has been removed permanently.

Cleaning the Data(4/7)

50

LESSON

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**

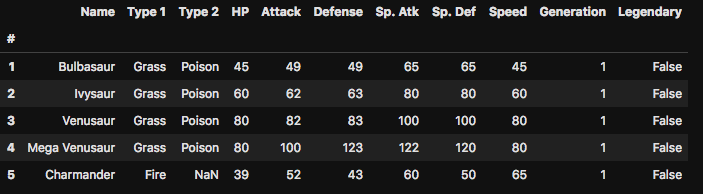
Another operation which although is not covered in any of the tasks here but you would be frequently used while dealing with data is the .reset\_index() method. In the image below in the first code snippet where we set the index of the dataframe according to the values in '#' column.

*# Set '#' as index*

df\_2 = df.set\_index('#')

df\_2.head()

**Output**



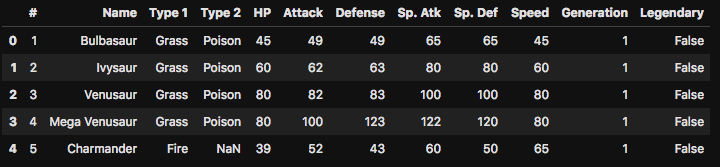
Now imagine you need to revert it to the original index form due to some reasons. You can do it with the reset\_index() method. This method will simply push the index values into a column and set default values as an index.

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()



Exploring Categorical Data(5/7)

50

LESSON

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(6/7)

50

LESSON

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