QA
POWERING
POTENTIAL

Python Libraries: Pandas

OVERVIEW

- What is Pandas
- Series
- Data Frame
- Group By
- Join
- Missing Values

WHAT IS PANDAS

- Pandas is Python's ETL package for structured data
- Built on top of NumPy, designed to mimic the functionality of R data frames
- Provides a convenient way to handle tabular data
- Can perform all SQL functionalities, including group-by and join
- Compatible with many other Data Science packages, including visualisation packages such as Matplotlib and Seaborn
- Defines two main data types:
  - → **pandas.Series**
  - → **pandas.DataFrame**

SERIES

- A **series** is a one-dimensional array that holding data of any type. It can be viewed as a column in a table.
- It consists of two NumPy arrays:
  - → **Index array**: stores the indices of the elements
  - → **Values array**: stores the values of the elements
- Each array element has a unique index (ID), contained in a separate index array.
- If we reorder the series, the index moves with the element, so an index will always identify with the same element in the series.
- Indices do not have to be sequential; they do not even have to be numbers.
- Think of indices as being the primary keys for each row in a single column table.

CREATING SERIES

A **Pandas series** can be created from a Python list or a NumPy array:

```
import pandas as pd

X = [1, 3, 5, 7]
mySeries = pd.Series(X)
print(mySeries)
```

```
0    1
1    3
2    5
3    7
dtype: int64
```

Index array

values array

→By default, the index starts from 0, and increments by 1 for each subsequent element in the series.

→The index is used to access the corresponding value.

```
print(mySeries[1])
```

```
3
```

CREATING SERIES

→ The index can be changed to another list or NumPy array of the same length:

```
mySeries.index = ["t","x","y","z"]
```

→ The index can be set at the time of its creating:

```
import pandas as pd

X = [1, 3, 5, 7]
mySeries = pd.Series(X,index=["a","b","c","d"])
print(mySeries)
```

```
a    1
b    3
c    5
d    7
dtype: int64
```

```
print(mySeries["c"])
```

```
5
```

QUERYING SERIES

Series can be used like an array, except the indices must correspond to the elements in the index array

series.index returns the index array

series.values returns the values array

series[ind] is equivalent to **series.loc[ind]**, returns the element in the series with ID equal to ind

series.iloc[i] returns the i-th element in the series

QUERYING SERIES

```
import pandas as pd

X = [1, 3, 5, 7]
mySeries = pd.Series(X,index=[9, 8, 7, 6])
print(mySeries)
```

```
9    1
8    3
7    5
6    7
dtype: int64
```

```
mySeries.index
```

```
Int64Index([9, 8, 7, 6], dtype='int64')
```

```
mySeries.values
```

```
array([1, 3, 5, 7], dtype=int64)
```

```
mySeries.loc[7]
```

← Index 7

```
5
```

```
mySeries.iloc[0]
```

← Position 0

```
1
```

DATA FRAME

- A Pandas **Data Frame** represents a table, and it contains:
  - → **Data in form of rows and columns**
  - → **Row IDs (the index array, i.e., primary key)**
  - → **Column names (ID of the columns)**
- A Data Frame is equivalent to collection of Series with each Series representing a column
- The row indices by default start from 0 and increase by one for each subsequent row, but just like Series they can be changed to any collection of objects.
- Each row index uniquely identifies a particular row. If we reorder the rows, their indices go with them.

Data frames are the data structures most suitable for analytics – rows representing observations and columns representing attributes of different data types.

CREATING DATA FRAMES

- Creating from Python lists, or NumPy arrays:

```
data = {
    "age": [34, 42, 27],
    "height": [1.78, 1.82, 1.75],
    "weight": [75, 80, 70]
}
df = pd.DataFrame(data)
print(df)
```

```
   age  height  weight
0   34    1.78      75
1   42    1.82      80
2   27    1.75      70
```

→ Input using a dictionary with column names as keys and the corresponding column data as values

- Creating from CSV files:

pandas.read_csv(csv_file_name)

→ The first row is used for column names

QUERYING DATA FRAMES

Getting entire rows:

my_dataframe.loc[row_id]

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

Row with index 0

```
age       34.00
height     1.78
weight    75.00
Name: 0, dtype: float64
```

```
# use a list of indices
df.loc[[0,1]]
```

Rows with indices 0 and 1

	age	height	weight
0	34	1.78	75
1	42	1.82	80

QUERYING DATA FRAMES

Indices can be named:

```
df_1 = pd.DataFrame(data, index = ["ind1", "ind2", "ind3"])

print(df_1)
```

```
      age  height  weight
ind1   34    1.78      75
ind2   42    1.82      80
ind3   27    1.75      70
```

```
df_1.loc["ind2"]
```

Row with index “ind2”

```
age       42.00
height     1.82
weight    80.00
Name: ind2, dtype: float64
```

```
df_1.iloc[2]
```

Row with position 2

```
age       27.00
height     1.75
weight    70.00
Name: ind3, dtype: float64
```

QUERYING DATA FRAMES

Getting entire columns:

my_dataframe.loc[:, col_name]

my_dataframe.iloc[y:,col_position]

```
df_1.loc[:,"age"]
```

Column “age”

```
ind1    34
ind2    42
ind3    27
Name: age, dtype: int64
```

```
df_1.iloc[:,0]
```

Column with position 0

```
ind1    34
ind2    42
ind3    27
Name: age, dtype: int64
```

QUERYING DATA FRAMES

Getting individual elements from row and column IDs:

my_dataframe.loc[row_id, col_name]

my_dataframe.iloc[i, j]

```
df_1.loc["ind1","height"]
```

1.78

Row index “ind1”
Column “height”

```
df_1.iloc[0,1]
```

1.78

Row 0 Column 1

QUERYING DATA FRAMES

my_dataframe.loc[[id1, id2, id3], :]

returns rows id1, id2 and id3, all columns

my_dataframe.loc[:, [col1, col2, col3]]

returns columns col1, col2 and col3, all rows

my_dataframe.loc[[id1, id2, id3], [col1, col2, col3]]

returns 3 by 3 table of rows id1, id2 and id3, columns col1, col2, and col3

Group By

Group the table rows into sub-groups according to a given criteria.

DataFrame

Index	Name	Gender	Age
0	Alice	Female	23
1	Bob	Male	26
2	Charlie	Male	25
3	Dave	Male	24

my_dataframe

Series

Index	Group
0	A
1	B
2	A
3	B

criteria

Group	Name	Gender	Age
A	Alice	Female	23
A	Charlie	Male	25

Group	Name	Gender	Age
B	Bob	Male	26
B	Dave	Male	24

my_dataframe.groupby(criteria)

GROUP BY

GROUP BY and:

→ Counting the number of rows in each group:

my_dataframe.groupby(criteria).size()

→ Sum of every numerical column in each group:

my_dataframe.groupby(criteria).sum()

→ Mean of every numerical column in each group:

my_dataframe.groupby(criteria).mean()

Join

Index	Name	Gender	Age
0	Alice	Female	23
1	Dave	Male	24

Index	Name	Tel
0	Alice	+4478654345
1	Bob	+4471749834
2	Charlie	+4479973432

Outer join

Index	Name	Gender	Age	Tel
0	Alice	Female	23	+4478654345
1	Bob	Male	NaN	+4471749834
2	Charlie	Male	NaN	+4479973432
3	Dave	Male	24	NaN

Inner join

Index	Name	Gender	Age	Tel
0	Alice	Female	23	+4478654345

Left join

Index	Name	Gender	Age	Tel
0	Alice	Female	23	+4478654345
1	Dave	Male	24	NaN

Right join

Index	Name	Gender	Age	Tel
0	Alice	Female	23	+4478654345
1	Bob	Male	NaN	+4471749834
2	Charlie	Male	NaN	+4479973432

JOIN

Use **DataFrame.merge()** as a general method of joining two data frames:

```
dataframe_A.merge(dataframe_B, left_on = ‘Name’, right_on = ‘Name’, how = ‘inner’)
```

```
dataframe_A.merge(dataframe_B, left_on = ‘Name’, right_on = ‘Name’, how = ‘outer’)
```

```
dataframe_A.merge(dataframe_B, left_on = ‘Name’, right_on = ‘Name’, how = ‘left’)
```

```
dataframe_A.merge(dataframe_B, left_on = ‘Name’, right_on = ‘Name’, how = ‘right’)
```

MISSING VALUES

Missing values in Pandas are represented by the NumPy object: **numpy.nan**

- **numpy.nan** represents **NaN**, or “Not a Number”
- numpy.nan cannot participate in arithmetic or comparison operations. Any arithmetic or comparison operation involving numpy.nan will always return numpy.nan.

numpy.nan + 42 → result is **numpy.nan**

numpy.nan < 3 → result is **numpy.nan**

- **numpy.isnan()** is used to check if a variable is NaN
- **~numpy.isnan()** is used to check if a variable is not NaN

TREATING MISSING VALUES

- **Finding out the number of missing values in each column**

my_dataframe.isna().sum()

- **Removing the rows with missing values**

my_dataframe.dropna(axis = 0)

- **Removing the columns with missing values**

my_dataframe.dropna(axis = 1)

- **Filling with a value**

→**For all missing values:**
my_dataframe.fillna(replacement_value)

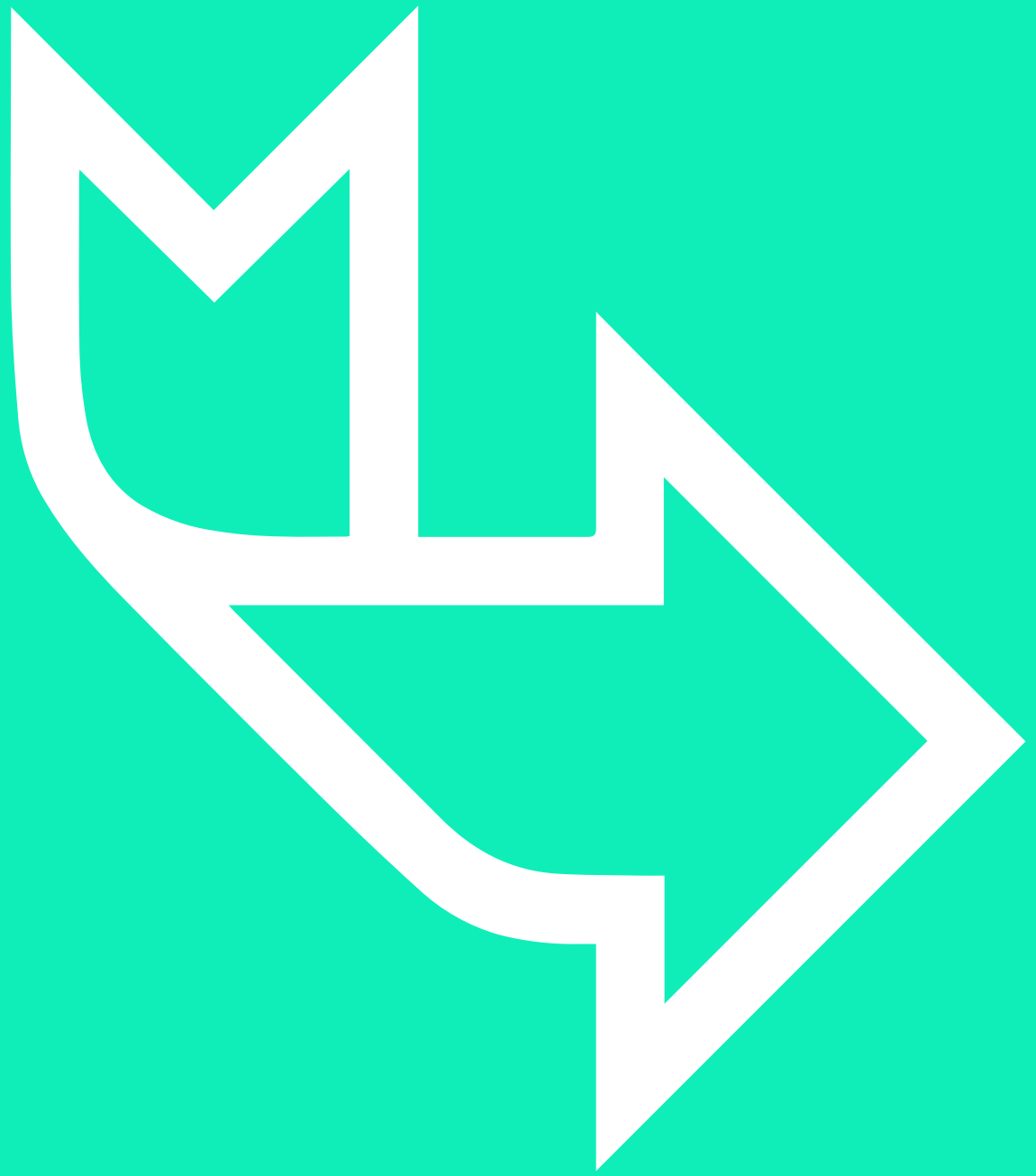
→**Different value for each column:**

my_dataframe.fillna({'NAME': 'UNKNOWN', 'AGE': 0})

NOTE

axis = 0 – rows

axis = 1 - columns



Further Reading

https://www.python.org/