DSP505: Programming Lab for Data Science and Artificial Intelligence

TPL616: Advanced Programming for DSAI

(Pandas Tutorial)

Vishwesh Jatala

Assistant Professor

Department of CSE

Indian Institute of Technology Bhilai

vishwesh@iitbhilai.ac.in

Acknowledgement

Today's lecture are borrowed from:

http://jake-feldman.squarespace.com/data-science-python-oscm400c

Why Pandas ?

- "Pandas" is a contraction of the words "Panel" and "Data," but it is also a contraction of the term "Python Data Analysis."
- Popular data science tool
- Rich relation data tool built on the top of Numpy
- High Performance and better than other open source tools

Pandas Features

- Two data structures
  - Series
  - Dataframes

- Read data from various file formats: CSV, Excel, JSON, SQL

- Statistics: Filter and aggregate data

- Data manipulation

Pandas Series

- A Pandas Series is a one-dimensional labeled array that can hold data of any type (integers, floats, strings, objects, etc.).

- A series is a like a single column; used for single variable analysis

```
import pandas as pd


# From a list
s1 = pd.Series([10, 20, 30, 40])
```

Output:

```
0     10
1     20
2     30
3     40
dtype: int64
```

Pandas Series

- Panda’s series can be created from Numpy, Tuples, Lists, Dictionaries and Tuples.

```python
import numpy as np
import pandas as pd

# 1. From NumPy Array
arr = np.array([10, 20, 30, 40])
s1 = pd.Series(arr)
print("Series from NumPy Array:")
print(s1, "\n")

# 2. From Tuple
tup = (100, 200, 300, 400)
s2 = pd.Series(tup, index=['a',
'b', 'c', 'd'])
print("Series from Tuple:")
print(s2, "\n")
```

```python
import numpy as np
import pandas as pd


# 3. From Dictionary
data = {'x': 1, 'y': 2, 'z': 3}
s3 = pd.Series(data)
print("Series from Dictionary:")
print(s3)
```

Pandas DataFrame

- Dataframe is a 2D table with rows and columns.

```
import pandas as pd

# Dictionary data
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'City': ['New York', 'London',
'Paris']
}
# Create DataFrame
df = pd.DataFrame(data)
print(df)
```

Output:

```
      Name  Age      City
0    Alice   25  New York
1      Bob   30    London
2  Charlie   35     Paris
```

Reading From Data Files

- Pandas supports reading data from various file formats: CSV/EXCEL/JSON/SQL

```
import pandas as pd
import numpy as np

df_csv = pd.read_csv("data.csv")

df_excel = pd.read_excel("data.xlsx", sheet_name="Sheet1")

df_json = pd.read_json("data.json")
```

- For storing the data:
  - df.to_csv()
  - df.to_excel()
  - df.to_json()

Optimization for Data Loading

- Efficient data loading is critical for improving performance, especially for large datasets.
  - usecols: Read only required columns
  - dtype: Reduce memory size
  - chunksize: Read in chunks
  - compression: Read compressed files directly

```python
import pandas as pd


df_usecols = pd.read_csv("data.csv", usecols=["Name", "Age"])
print("Only selected columns:\n", df_usecols.head(), "\n")


dtype_dict = {"Age": "int8"}
df_dtype = pd.read_csv("data.csv", dtype=dtype_dict)
print("With optimized dtypes:\n", df_dtype.dtypes, "\n")
```

Optimization for Data Loading

- Efficient data loading is critical for improving performance, especially for large datasets.
  - usecols: Read only required columns
  - dtype: Reduce memory size
  - chunksize: Read in chunks
  - compression: Read compressed files directly

```
chunks = pd.read_csv("data.csv", chunksize=2)
print("Processing in chunks:")
for chunk in chunks:
   print(chunk, "\n")


df_usecols.to_csv("data.csv.gz", index=False,compression="gzip"
df_gzip = pd.read_csv("data.csv.gz", compression="gzip")
print("Loaded from compressed gzip file:\n", df_gzip, "\n")
```

Accessing the Loaded Data

- Basic features
- head()
- Accessing individual elements
- Slicing series
- Slicing dataframes

The head() Method

Using the **head()** method

```
import pandas as pd

df_grades = pd.read_csv("Grades_Short.csv")
df_grades.head(3)
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-

- If the data is really large you don't want to print out the entire dataframe to your output.

- The **head(n)** method outputs the first n rows of the data frame. If n is not supplied, the default is the first 5 rows.

- I like to run the head() method after I read in the dataframe to check that everything got read in correctly.

- There is also a **tail(n)** method that returns the last n rows of the dataframe

Basic Features

```
import pandas as pd

df_grades = pd.read_csv("Grades_Short.csv")
df_grades.head(3)
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-

```
#dimension of df
df_grades.shape
```

```
(7, 9)
```

Think of this as a list

```
#How each column is stored
df_grades.dtypes
```

```
Name               object
Previous_Part     float64
Participation1      int64
Mini_Exam1        float64
Mini_Exam2          int64
Participation2      int64
Mini_Exam3        float64
Final             float64
Grade              object
dtype: object
```

object = string

float64 = decimal

int64 = integer

Basic Features

column names

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-

row names = index

Selecting a Single Column

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

```
#Get Name column
df_grades['Name']
```

```
0      Jake
1       Joe
2     Susan
3       Sol
4     Chris
5     Tarik
6     Malik
Name: Name, dtype: object
```

- Between square brackets, the column must be given as a string
- Outputs column as a series
    - A series is a one dimensional dataframe..more on this in the slicing section

Selecting Multiple Columns

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

```
#Select multiple columns
df_grades[["Name", "Grade"]]
```

	Name	Grade
0	Jake	A
1	Joe	A
2	Susan	A-
3	Sol	A
4	Chris	A
5	Tarik	B
6	Malik	A

- List of strings, which correspond to column names.
- You can select as many column as you want.
- Column don’t have to be contiguous.

Slicing a Series

Slice/index through the index, which is usually numbers

```
names= df_grades["Name"]
names
```

```
0     Jake
1      Joe
2    Susan
3      Sol
4    Chris
5    Tarik
6    Malik
Name: Name, dtype: object
```

Slicing a Series

```
names= df_grades["Name"]
names
```

```
0     Jake
1      Joe
2    Susan
3      Sol
4    Chris
5    Tarik
6    Malik
Name: Name, dtype: object
```

Slice/index through the index, which is usually numbers

Picking out single element

```
names[0]
```

```
'Jake'
```

Slicing a Series

```
names= df_grades["Name"]
names
```

```
0     Jake
1      Joe
2    Susan
3      Sol
4    Chris
5    Tarik
6    Malik
Name: Name, dtype: object
```

Slice/index through the index, which is usually numbers

Picking out single element

```
names[0]
```

```
'Jake'
```

Contiguous slice

non_inclusive

```
names[1:4]
```

```
1      Joe
2    Susan
3      Sol
Name: Name, dtype: object
```

Slicing a Series

```
names= df_grades["Name"]
names
```

```
0       Jake
1        Joe
2      Susan
3        Sol
4      Chris
5      Tarik
6      Malik
Name: Name, dtype: object
```

Slice/index through the index, which is usually numbers

Picking out single element

```
names[0]
```

```
'Jake'
```

Contiguous slice

```
names[1:4]
```

```
1        Joe
2      Susan
3        Sol
Name: Name, dtype: object
```

Arbitrary slice

```
names[[1,2,4]]
```

```
1        Joe
2      Susan
4      Chris
Name: Name, dtype: object
```

Slicing a Data Frame

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

- There are a few ways to pick slice a data frame, we will use the .loc method.
- Access elements through the index labels column names
    - We will see how to change both of these labels later on

Slicing a Data Frame

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

- Pick a single value out.

Index label (number)

Column name (string)

```
first_name = df_grades.loc[0,"Name"]
first_name
```

```
'Jake'
```

Slicing a Data Frame

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

- Pick out entire row:

```
first_row = df_grades.loc[0,:]
first_row
```

“pick out all columns”

```
Name              Jake
Previous_Part       32
Participation1       1
Mini_Exam1        19.5
Mini_Exam2          20
Participation2       1
Mini_Exam3          10
Final               33
Grade                A
Name: 0, dtype: object
```

first_row is a series

Slicing a Data Frame

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

- Pick out contiguous chunk:

Endpoints are inclusive!

```
slice_one = df_grades.loc[0:2,"Name":"Mini_Exam2"]
slice_one
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2
0	Jake	32.0	1	19.5	20
1	Joe	32.0	1	20.0	16
2	Susan	30.0	1	19.0	19

Slicing a Data Frame

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

- Pick out arbitrary chunk:

```
slice_two = df_grades.loc[[0,2,3], ["Name", "Grade"]]
slice_two
```

	Name	Grade
0	Jake	A
2	Susan	A-
3	Sol	A

Built in Functions

```python
import pandas as pd

df_grades = pd.read_csv("Data/Grades_Short.csv")
df_grades
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

How do I compute the average score on the final?

Built in Functions

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

How do I compute the average score on the final?

```
#Print out
df_grades.Final.mean()
```

```
32.214285714285715
```

```
#Store
avg_final = df_grades.Final.mean()
avg_final
```

```
32.214285714285715
```

Built in mean() method

Built in Functions

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

How do I compute the highest Mini Exam 1 score?

```
max_mini_1 = df_grades["Mini_Exam1"].max()
max_mini_1
```

```
22.0
```

Built in Functions

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

I can actually get all key stats for *numeric* columns at once with the describe() method:

```
summary_df = df_grades.describe()
summary_df
```

summary_df is a dataframe!

	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final
count	7.000000	7.0	7.000000	7.000000	7.0	7.000000	7.000000
mean	31.071429	1.0	19.785714	17.857143	1.0	11.000000	32.214286
std	0.838082	0.0	1.074598	2.734262	0.0	2.217356	3.828154
min	30.000000	1.0	19.000000	13.000000	1.0	8.000000	24.000000
25%	30.500000	1.0	19.000000	16.500000	1.0	9.500000	32.500000
50%	31.000000	1.0	19.500000	19.000000	1.0	10.500000	33.000000
75%	31.750000	1.0	20.000000	19.500000	1.0	12.750000	33.750000
max	32.000000	1.0	22.000000	21.000000	1.0	14.000000	36.000000

Built in Functions

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

I can actually get all key stats for *numeric* columns at once with the describe() method:

```
summary_df = df_grades.describe()
summary_df[["Final", "Mini_Exam3"]]
```

	Final	Mini_Exam3
count	7.000000	7.000000
mean	32.214286	11.000000
std	3.828154	2.217356
min	24.000000	8.000000
25%	32.500000	9.500000
50%	33.000000	10.500000
75%	33.750000	12.750000
max	36.000000	14.000000

Built in Functions

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

I can actually get all key stats for *numeric* columns at once with the describe() method:

```
summary_df = df_grades.describe()
summary_df[["Final", "Mini_Exam3"]]
```

	Final	Mini_Exam3
count	7.000000	7.000000
mean	32.214286	11.000000
std	3.828154	2.217356
min	24.000000	8.000000
25%	32.500000	9.500000
50%	33.000000	10.500000
75%	33.750000	12.750000
max	36.000000	14.000000

Notice here the index is *not* row numbers…

Built in Functions

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

Other useful built in methods:

```
df_grades["Grade"].value_counts()
```

```
A      5
A-     1
B      1
Name: Grade, dtype: int64
```

value_count(): Gives a count of the number of times each unique value apears in the column. Returns a series where indices are the unique column values.

Built in Functions

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

Other useful built in methods:

```
counts = df_grades["Grade"].value_counts()
counts
```

```
A     5
A-    1
B     1
Name: Grade, dtype: int64
```

```
counts["A"]
```

```
5
```

value_count(): Gives a count of the number of times each unique value appears in the column. Returns a series where indices are the unique column values.

Built in Functions

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

Other useful built in methods:

```
df_grades["Grade"].unique()
```

```
array(['A', 'A-', 'B'], dtype=object)
```

```
unique_values = df_grades["Grade"].unique()
unique_values[0]
```

```
'A'
```

```
len(unique_values)
```

```
3
```

unique(): Returns an array of all of the unique values.

Missing Data

- Missing data is common in data science.
- Need to be handled to avoid bias.
- Techniques
  - Drop values
  - Impute them with substitutes

Missing Data

A	B	C	D	E	F	G	H	I	J
Name	Previous_Pai	Participation	Mini_Exam1	Mini_Exam2	Participation	Mini_Exam3	Final	Grade	Temp
Jake	32	1	19.5	20	1	10	33	A	-1
Joe	NA	1	20	16	1	14	32	A	23
Sol	31	1	22	13	1	13	34	A	34
Chris	30	-1	19	not available	1	12.5	33.5	A	72

```
df_missing = pd.read_csv("Data/Missing_Data.csv")
df_missing
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	Temp
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A	-1
1	Joe	NaN	1	20.0	16	1	14.0	32.0	A	23
2	Sol	31.0	1	22.0	13	1	13.0	34.0	A	34
3	Chris	30.0	-1	19.0	not available	1	12.5	33.5	A	72

Not that different columns have different indicators for missing data.

Missing Data

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	Temp
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A	-1
1	Joe	NaN	1	20.0	16	1	14.0	32.0	A	23
2	Sol	31.0	1	22.0	13	1	13.0	34.0	A	34
3	Chris	30.0	-1	19.0	not available	1	12.5	33.5	A	72

```
df_missing.dtypes
```

```
Name                object
Previous_Part      float64
Participation1       int64
Mini_Exam1         float64
Mini_Exam2          object
Participation2       int64
Mini_Exam3         float64
Final              float64
Grade               object
Temp                 int64
dtype: object
```

We can replace the missing data with a true NaN (right now everything is just a string).

Missing Data

```
df_missing = pd.read_csv("Data/Missing_Data.csv", \
                          na_values=["NaN", "not available"])
df_missing
```

List of strings specifying which values are missing.

Missing Data

```
df_missing = pd.read_csv("Data/Missing_Data.csv", \
                         na_values=["NaN", "not available"])
df_missing
```

List of strings specifying which values are missing.

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	Temp
0	Jake	32.0	1	19.5	20.0	1	10.0	33.0	A	-1
1	Joe	NaN	1	20.0	16.0	1	14.0	32.0	A	23
2	Sol	31.0	1	22.0	13.0	1	13.0	34.0	A	34
3	Chris	30.0	-1	19.0	NaN	1	12.5	33.5	A	72

Special NaN value (from numpy package), which is not a string.

Missing Data

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	Temp
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A	-1
1	Joe	NaN	1	20.0	16	1	14.0	32.0	A	23
2	Sol	31.0	1	22.0	13	1	13.0	34.0	A	34
3	Chris	30.0	-1	19.0	not available	1	12.5	33.5	A	72

We know “NaN” and “not available” are missing data points, but what about -1?

Missing Data

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	Temp
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A	-1
1	Joe	NaN	1	20.0	16	1	14.0	32.0	A	23
2	Sol	31.0	1	22.0	13	1	13.0	34.0	A	34
3	Chris	30.0	-1	19.0	not available	1	12.5	33.5	A	72

We know "NaN" and "not available" are missing data points, but what about -1?

- For the Participation1 column the -1 is probably missing data.
- For the Temp column, the -1 is likely not missing data, since -1 is a valid temperature.

For each column, we can specify exactly which values correspond to missing data.

Missing Data

```
df_missing = pd.read_csv("Data/Missing_Data.csv", na_values={"Mini_Exam2" : "not available",\
                                                             "Participation1": -1})
df_missing
```

Curly brackets

Missing Data

```
df_missing = pd.read_csv("Data/Missing_Data.csv", na_values={"Mini_Exam2" : "not available",\
                                                              "Participation1": -1})

df_missing
```

Column name as string

NaN value

Missing Data

```
df_missing = pd.read_csv("Data/Missing_Data.csv", na_values={"Mini_Exam2" : "not available",\
                                                               "Participation1": -1})
df_missing
```

Column name as string

NaN value

“For column Participation1, replace all -1s with a NaN.”

Missing Data

```
df_missing = pd.read_csv("Data/Missing_Data.csv", na_values={"Mini_Exam2" : "not available",\
                                                             "Participation1": -1})
df_missing
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	Temp
0	Jake	32.0	1.0	19.5	20.0	1	10.0	33.0	A	-1
1	Joe	NaN	1.0	20.0	16.0	1	14.0	32.0	A	23
2	Sol	31.0	1.0	22.0	13.0	1	13.0	34.0	A	34
3	Chris	30.0	NaN	19.0	NaN	1	12.5	33.5	A	72

Notice that the -1 was replaced only in Participation1 column

Benefiting of Having NaNs

- Have common symbol for where there is missing data
  - Good for you and good for others looking at your code/data
  - These entries will be ignored if you try to compute means of columns with NaNs.
- We can easily get rid of column/rows with missing data
- We can easily replace the missing values with the mean of the column, for example.

Dropna() Method

```
df_missing = pd.read_csv("Data/Missing_Data.csv", na_values=["NaN", "not available",\
                                                              -1])
df_missing
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	Temp
0	Jake	32.0	1.0	19.5	20.0	1	10.0	33.0	A	NaN
1	Joe	NaN	1.0	20.0	16.0	1	14.0	32.0	A	23.0
2	Sol	31.0	1.0	22.0	13.0	1	13.0	34.0	A	34.0
3	Chris	30.0	NaN	19.0	NaN	1	12.5	33.5	A	72.0

How do I get rid of all rows with NaN?

Dropna() Method

```
df_missing = pd.read_csv("Data/Missing_Data.csv", na_values=["NaN", "not available",\
                                                              -1])
df_missing
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	Temp
0	Jake	32.0	1.0	19.5	20.0	1	10.0	33.0	A	NaN
1	Joe	NaN	1.0	20.0	16.0	1	14.0	32.0	A	23.0
2	Sol	31.0	1.0	22.0	13.0	1	13.0	34.0	A	34.0
3	Chris	30.0	NaN	19.0	NaN	1	12.5	33.5	A	72.0

How do I get rid of all rows with NaN?

```
df_missing.dropna(axis = 0, inplace=False)
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	Temp
2	Sol	31.0	1.0	22.0	13.0	1	13.0	34.0	A	34.0

- Setting axis = 1 would drop all columns with an NaN

Drop rows where all values are missing
df_drop_all = df.dropna(how='all')

Missing Data

- When to drop?
  - Missing data is not critical
  - Less than 5% of the data
  - Dropping the data does not introduce bias

Missing Data

- Missing data is common in data science.
- Need to handled to avoid bias.
- Techniques
  - Drop values
  - Impute them with substitutes

Imputing Missing Values

- **Constant Values**

- **Mean / Median / Mode**:
  - Mean if data is normally distributed.
  - Median if there are outliers.
  - Mode for categorical values.

- **Forward fill / Backward fill** → Use for time series data to carry forward or backward the nearest known value.

- **Interpolation** → Use for continuous data like sensors or finance, where trends matter.

- **ML-based (KNN, Iterative)** → Use for complex datasets where simple methods are not accurate enough.

Fillna() Method

```
df_missing = pd.read_csv("Data/Missing_Data.csv", na_values=["NaN", "not available",\
                                                              -1])
df_missing
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	Temp
0	Jake	32.0	1.0	19.5	20.0	1	10.0	33.0	A	NaN
1	Joe	NaN	1.0	20.0	16.0	1	14.0	32.0	A	23.0
2	Sol	31.0	1.0	22.0	13.0	1	13.0	34.0	A	34.0
3	Chris	30.0	NaN	19.0	NaN	1	12.5	33.5	A	72.0

Rather than getting rid of rows/columns, we fill the “holes” in a number of ways.

```
#Replace with specific value
df_missing.fillna(0, inplace=False)
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	Temp
0	Jake	32.0	1.0	19.5	20.0	1	10.0	33.0	A	0.0
1	Joe	0.0	1.0	20.0	16.0	1	14.0	32.0	A	23.0
2	Sol	31.0	1.0	22.0	13.0	1	13.0	34.0	A	34.0
3	Chris	30.0	0.0	19.0	0.0	1	12.5	33.5	A	72.0

Fillna() Method

```
df_missing = pd.read_csv("Data/Missing_Data.csv", na_values=["NaN", "not available",\
                                                              -1])
df_missing
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	Temp
0	Jake	32.0	1.0	19.5	20.0	1	10.0	33.0	A	NaN
1	Joe	NaN	1.0	20.0	16.0	1	14.0	32.0	A	23.0
2	Sol	31.0	1.0	22.0	13.0	1	13.0	34.0	A	34.0
3	Chris	30.0	NaN	19.0	NaN	1	12.5	33.5	A	72.0

Rather than getting rid of rows/columns, we fill the “holes” in a number of ways.

```
#Replace with specific value in specific column
mean_temp = df_missing.Temp.mean()
df_missing.fillna({'Temp': mean_temp}, inplace=False)
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	Temp
0	Jake	32.0	1.0	19.5	20.0	1	10.0	33.0	A	43.0
1	Joe	NaN	1.0	20.0	16.0	1	14.0	32.0	A	23.0
2	Sol	31.0	1.0	22.0	13.0	1	13.0	34.0	A	34.0
3	Chris	30.0	NaN	19.0	NaN	1	12.5	33.5	A	72.0

Other Methods for Imputing

```
df = pd.DataFrame({
    "Name": ["Alice", "Bob", "Charlie", "David", "Eva"],
    "Age": [25, np.nan, 30, np.nan, 40],
    "City": ["NY", "London", np.nan, "Paris", np.nan]
})

print("Original Data:")
print(df, "\n")

ffill = df.fillna(method="ffill")
bfill = df.fillna(method="bfill")

interp = df.copy()
interp["Age"] = interp["Age"].interpolate(method="linear")
```

Missing Data

- When to impute?
  - When data loss is significant (more than 5%)
  - Missing values can be estimated easily
  - Domain knowledge can help in filling the missing data.

Outlier Detection

- An outlier is a data point that is very different (much higher or lower) from the rest of the dataset.
- Outlier detection is crucial for improving the data quality.
- Methods:
  - IQR (Interquartile Range)
  - Z-score

Outlier Detection

- IQR (Interquartile Range)
  - Q1: 25th Percentile
  - Q2: Median
  - Q3: 75t Percentile
  - IQR=Q3-Q1

- Outliers: Values outside 1.5 * IQR from Q1 and Q3.

Outlier Detection

- Z-score
    - $Z=(X-\mu)/\sigma$
- Outlier: Z-score > 3 (far from the mean)

Other Methods for Imputing

```
import pandas as pd

df = pd.DataFrame({'values': [1, 2, 3, 100, 4, 5, 200]})
mean_val = df['values'].mean()
std_val = df['values'].std()
df['z_score'] = (df['values'] - mean_val) / std_val
df['outlier_z'] = df['z_score'].apply(lambda x: abs(x) > 3)

Q1 = df['values'].quantile(0.25)
Q3 = df['values'].quantile(0.75)
IQR = Q3 - Q1
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

df['outlier_iqr'] = (df['values'] < lower_bound) |
(df['values'] > upper_bound)
```

Pipelines

- A pipeline is a way to chain multiple data transformations together so your code becomes:
  - Cleaner
  - More readable
  - Easier to debug
- .pipe() method

Other Methods for Imputing

```
import pandas as pd

# Sample Data
data = {
    'name': ['Alice', 'Bob', 'Charlie', 'David'],
    'age': [25, 17, 35, 45],
    'salary': [500, 600, 700, 800]
}
df = pd.DataFrame(data)

def filter_adults(data):
    return data[data['age'] > 18]

def add_salary_inr(data):
    data['salary_inr'] = data['salary'] * 80
    return data
```

Other Methods for Imputing

```
def avg_salary(data):
    return data['salary_inr'].mean()

# Apply pipeline
result = (df
         .pipe(filter_adults)
         .pipe(add_salary_inr)
         .pipe(avg_salary)
        )

print("Average Salary (INR):", result)
```

Summary

Pandas Cheatsheet

Imports

```
import pandas as pd
import altair as alt
import datetime
```

Creating DataFrames

```
df1 = pd.read_csv(  # From file
    'world_countries.csv')
df2 = pd.DataFrame({  # From Python dict
  'col0': [0, 1, 2],  'col1': [3, 4, 5],
  'col2': ['ab', 'cd', 'ef'],
  'col3': [datetime.datetime.now()] * 3})
```

Inspecting DataFrames

```
df1.head()       # First 5 rows
df1.tail()       # Last 5 rows
df1.columns      # Columns names
len(df1)         # Number of rows
df1.shape        # Number of rows and cols
df1.describe() # Stats about each column
df1.info()       # Summary info
```

Summarizing columns

```
# Rename a column
df1 = df1.rename(
  columns={'Population': 'Pop'})
df1.Pop.sum()      # Sum
df1.Pop.mean()     # Average
df1.Pop.std()      # Standard deviation
df1.Pop.median() # Median
df1.Pop.min()      # Minimum
df1.Pop.max()      # Maximum
```

Filtering rows

```
df1[5:11]       # Select rows 5 through 10
# Rows with Spain in the Country column
df1[df1.Country == "Spain"]
# Removing nulls
df1 = df1[~df1.Pop.isnull()]
# Convert strings to integers
df1.ConSal = df1.Pop.astype('int64')
# Booleans operators are &, | and ~
df1[(df1.Pop > 100) &
    ~(df1.Area.isnull())]
```

Column manipulations

```
# Arithmetic operations on columns
df2['col0'] + df2['col1']
# Even if they're strings
df2['col2'] + df2['col2']
# Create new column from the result
df2['col4'] = df2['col0'] / df2['col1']
# String methods and attributes can be
# accessed via .str.
df2['col2'].str.replace('a', 'b')
# And datetime methods and attributes
# via .dt.
df2.col3.dt.date
# Select just some columns from DataFrame
df1[['Country', 'Pop']]
```

Dealing with missing values

```
# Drop rows with any missing values
df1.dropna()
# Drop columns with any missing value
df1.dropna(axis=1)
# Fill missing values with 0s
df1.fillna(0)
# Fill missing values with ''
df1.fillna('')
```

Grouping

```
# Get the average salary for each country
df1.groupby('Country').agg(
    {'Pop': 'mean'})
# Get the average and minimum salary
df1.groupby('Country').agg(
    {'Pop': ['mean', 'min']})
# Keep grouping column as a column
df1.groupby(
    'Country', as_index=False).agg(
    {'Pop': ['mean', 'min']})
```

Miscellaneous

```
# Reorder from top salary to lowest
df1.sort_values('Pop',
                ascending=False)
# Remove a column
df1.drop(columns='Phones')
# Randomly select a sample of 45 rows
df1 = df1.sample(45)
```

Merging

```
df3 = pd.DataFrame(
    {'col5': ['ab', 'cd', 'ef'],
     'col6': [100, 200, 300]})
# Create a new DataFrame matching col2
# of df2 and col5 of df3.
df2.merge(df3, left_on='col2',
          right_on='col5')
```

Graphing

```
alt.Chart(df1).mark_point().encode(
  x='Country', y='Area', size='Pop',
  color='Birthrate')
```