(32) # pandas

↓

python module for data manipulation

↑

allows us to work with the data structured in row, column format

↓

cleaning data . add rows , add columns. remove them ,

↓

making data more accessible or in other words, operation ready

↓

managing data in a way similar to excel .

```
import pandas as pd
pd.read_csv()
```

→ columns seperated by commas

to load data

path of the csv file

↘ local device as well

↘ URL

similarly read_json()
read_html()
read_excel()
functions

example , iris = pd.read_csv( )

iris

↓

the pandas module, takes the first row as header automatically

↓

since iris csv did not have proper header row

↓

the 1st data row became header .

and pandas reported 149 rows x 5 columns instead

* iris csv had 150 rows x 5 columns

type (iris) (return type of read_csv)

↳ pandas dataframe

↓

2D data holder with certain rows and columns

* df = iris (assigns same pointer, changes made using one will reflect in the other as well)

assigns reference to the variable

↓

shallow copy

df = iris.copy()

↳ relates df to a new copy made of iris and referenced using df

also have a corresponding df.tail()

* df.head() → brief look of the data set

↓

prints few starting element of the dataset

↓

allows us to do basic analysis of the dataset wrt types and values.

argument sends how many elements we want to see

↓

by default = 5

* df.shape → (the resolution of the dataframe i.e no. of rows and no. of columns in total)

↓

df.shape

* df.columns → to look at current headers

↓

to change headers, we need to send in a list of apt. size.

↓

df.columns = ['sl', 'sw', 'pl', 'pw', 'flower.type']

* lost the row which was used for columns

* df.dtypes
  ↓
  returns the type of values stored in each column
  ↓
  i.e. sl : float 64
  sw : float 64
  pl : float 64
  pw : float 64
  flower type : object

* df.describe()
  ↳ describes our data frame column wise
  ↓
  i) count ← valid no of values in column (ignores NAN)
  ii) mean
  iii) std — standard deviation
  iv) min
  v) 25%.
  vi) 50%.
  vii) 75%.
  viii) max

  Not valid value

  spread of the data

  percentile
  ↓
  value having 25% values less than it

to access a column of data frame

```
df.sl
df["sl"]
```

* df.isnull()
  ↳ wherever null values are present, it returns true otherwise false elementwise in the data

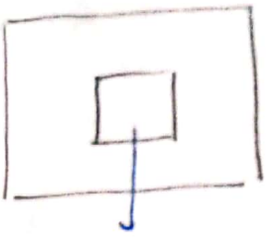
* df.isnull().sum()
  ↳ sum columnwise

↗ accessing part of data → slice of the data.

* df.iloc [1:4 , 2:3]

rows 1–3,
cols 2

	pl
1	1.3
2	1.5
3	2.4



to select a part of our data.
↓
direct slicing is not allowed.

* scikit → modelling data
↓
not on loading, manipulating and summarizing data features

* to delete rows in dataframe
↓
df.drop(0) → label of the row to be excluded as argument

makes a copy without the specified row
↓
to delete,

df = df.drop(0)
or
df.drop(0, inplace = TRUE)
↓
(by default, inplace is false)

indexing doesn't change automatically
↓
difference b/w label and position

(•) df.index → to look at labels of the dataframe
↓
df.index[0]
↳ gives label of element / row with position 0

# to drop by position
df.drop(df.index[0])

* to drop multiple rows at once, use comma

↓

df.drop(df.index[[0,1]], inplace = True)

↓

gives list of labels to drop

* df.sl(>)5

↓

runs condition on column and returns a set of bools based on the condition

↓

(returns a column)

↳ like a boolean array

* df[df.sl>5]

↓

returns complete rows of dataframe in which the specified condition, in this case the column sl of df satisfied.

3 different kinds of flowers in our data set Iris

a) Iris-setosa
b) Iris-versicolor
c) Iris-virginica

↓

df[df.flower_type=='Iris-setosa']

↓

df[df.flower_type=='Iris-setosa'].describe()

↓

provides interesting insights based on dataframe selected using describe function of pandas

↓

mean of different flower type comparison by storing in different classes

#I add a row

* df.iloc() → position based ← index based

* inserting using position
* if already present

df.loc()
↳ label based

df.loc[0] = [1, 2, 3, 4, "Iris-setosa"]

↓

adds a row at the end, with label 0

↓

Using iloc, we can insert on position basis

↓

overwrites if already present.

* based on dropping operations or data manipulation operations, the dataframes may have unordered indices / labels, which may cause referencing inconsistency.

↓

df.reset_index()

↓

* not making changes in df
* giving a new data frame

↓

to place changes in the same dataframe

↓

df.reset_index(drop = True, inplace = True)

↳ adds a new column with old labels named index by putting in indices reseted to 0.

↓

avoid adding that by adding a drop argument

↓

df.reset_index(drop = True)

df.index → describes the label of the dataframe

(38) # deleting columns in dataframe

df.drop('sl', axis = 1) ↘ drops 'sl' column

* axis → in arguments refers to the axis about which we operate in ND arrays.

↓

think of it as the coordinate axis,

0 → x axis

1 → y axis

2 → z axis

⋮

(based on index)

axis = 1 refers to column space

df.drop('sl', axis = 1, inplace = True)

* alternate method to drop

del df['sl']

↓

Will make changes in df itself

adding columns in dataframe

↓

for ex. if we want column with diff of pw and pl

↓

we insert using

↓

df["diff-pl-pw"] = df["pl"] - df["pw"]

similar to dictionary

df ["abc"] = 1

↳ column of all ones

Handling NAN

↓

we can do two things to handle NaN values. i.e.

(a) filling them with certain defined values.

(b) or removing them

↓

i.e. (a) drop na

(b) ~~remove~~ na
fill

*) trying indexing using iloc. to fetch certain indexed row

* inserting NaN in dataframe (for demonstration purposes)

df.iloc[2:4, 1:3] = np.nan → constant in numpy

↓

import numpy as np.

*) describe will ignore these values

* Handling methods.

(i) df.dropna() → drops rows with any one or more values = NaN.

↑ to drop entire columns with na, axis = 1.

↓

again df doesn't change without using inplace.

→ df.dropna(inplace = True)

↓

and then reset

(ii) filling na → this can be done in many ways i.e. different values can be inserted based on our requirements

mean of the specific column

↓

df[codes].sw.fillna(df[codes].sw.mean())

the column whose nan needs to be handled

* again not inplace *

df.sw.fillna(df.sw.mean(), inplace = True)

* to insert mean of that Flower class to maintain a sense of distribution

find the replacement class i.e

a = df [df.Flower_type == '_']

then

df.sw.fillna(a.sw.mean(), inplace = True)

Handling strings in data

↓

it is much more easier to operate on numerical data as compared to string data

↓

for example, the facebook or advertisement example of req. male and female data, and how easy it would be to perform computations on that data if numerical

↓

so we change ~~data~~ string based data to numeric data.

string based data

```
df["Gender"] = "Male"
df.iloc[0:10,c] = "Female"
def f
    df["sex"] = df.Gender.apply(f)
```

applying function f on the selected column and puts it into a new column by the name of sex

↓

and then delete gender column

returns a column object

```
def f(s):
    if s == "Male":
        return 0
    else:
        return 1
```

~~del df["gender"]~~

```
del df["Gender"]
```

This puts comparative association based on magnitude 0 and 1

← better to use one-hot encoding

(92) * Titanic Dataset

(i) useful columns.

(ii) Data processing requirements.

↓

useless columns drops
adding new columns.

* understanding what different features mean

↓

useful columns

to our final output

↓

drop any thing else

↓

adding new columns

analysing preprocessing required about the data.

→ Data cleaning and analysis — titanic dataset

* i) understanding meaning of each column

ii) analyse columns which can be deleted

iii) replace columns with string (if any) with int values for analysis

iv) Fill the null values of the age column

↓

Fill mean survived age (mean age of survived people)

in the column where the person has survived and mean not survived age (mean age of people who didn't survive)

v) draw graph for analysis