Lab Manual for Tools and Technologies for Data Science

Lab-03

Introduction to Pandas

Table of Contents

Lab 3: Introduction to Pandas

1. Objective

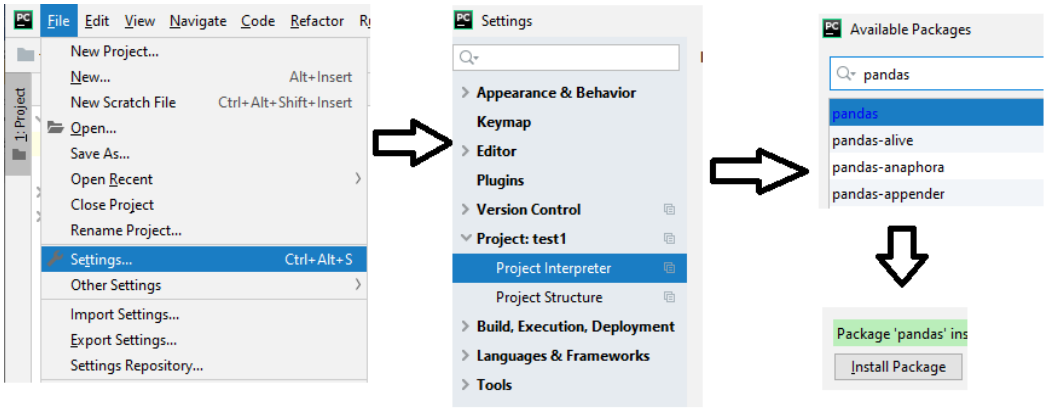
- Getting started with pandas
- Introduction to panda's data structure
- NumPy vs pandas
- Series vs Data frames
- Creating, Reading and Writing
- Indexing, Selecting and assigning data to data frames
- Grouping and sorting
- Handling missing values
- Use fundamental Python data frames for data science

2. Getting Started with pandas:

In the previous lab, we dove into detail on NumPy and its functionalities, which provides efficient storage and manipulation of dense typed arrays in Python. Here we'll build on this knowledge by looking in detail at the data structures provided by the Pandas library. Pandas is a newer package built on top of NumPy, and provides an efficient implementation of a Series and DataFrame. In this lab, we will focus on the mechanics of using Series, DataFrame, and related structures effectively. We will use examples drawn from real datasets where appropriate.

Installing and Using Pandas:

- ❖ Installing Pandas on your system requires NumPy to be installed
- ❖ Just follow the procedure to install package in PyCharm.



- ❖ Once Pandas is installed, you can import it and check the version:

```
import pandas
print(pandas.__version__)
```

Just as we generally import NumPy under the alias np, we will import Pandas under the alias pd:

```
import pandas as pd
```

This important convention will be used through the lab.

2.1 Introduction to pandas Data Structures

At the very basic level, Pandas objects can be thought of as enhanced versions of NumPy structured arrays in which the rows and columns are identified with labels rather than simple integer indices. To get started with pandas, you will need to get comfortable with its two workhorse data structures: Series and DataFrame. While they are not a universal solution for every problem, they provide a solid, easy-to-use basis for most applications.

2.2 Series

- A Pandas Series is a one-dimensional array of indexed data. It can be created from a list or array as follows:

Example:

```
data = pd.Series([0.25, 0.5, 0.75, 1.0])
print(data)
```

```
0    0.25
1    0.50
2    0.75
3    1.00
dtype: float64
```

Series wraps both a sequence of values and a sequence of indices, which we can access with the values and index attributes.

```
print(data.values)   //output: [0.25 0.5  0.75 1.  ]
```

Like with a NumPy array, data can be accessed by the associated index via the familiar Python square-bracket notation:

```
print(data[1])      //output: 0.5
```

2.2.1 Series as generalized NumPy array

From what we've seen so far, it may look like the Series object is basically interchangeable with a one-dimensional NumPy array. The essential difference is the presence of the index: while the NumPy array has an implicitly defined integer index used to access the values, the Pandas Series has an explicitly defined index associated with the values. This explicit index definition gives the Series object additional capabilities. For example, the index need not be an integer, but can consist of values of any desired type. For example, if we wish, we can use strings as an index:

```
data = pd.Series([0.25, 0.5, 0.75, 1.0], index=['a', 'b', 'c', 'd'])
print(data)
```

output:

```
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
```

and the item access works as expected: data[‘b’] = ?

Important: We can even use noncontiguous or nonsequential indices:

```
data = pd.Series([0.25, 0.5, 0.75, 1.0], index=[2, 5, 3, 7])
```

2.2.2 Series as specialized dictionary

We can make the Series-as-dictionary analogy even more clear by constructing a Series object directly from a Python dictionary:

```
population_dict = {'Lahore': 38332521,'Islamabad':
26448193,'Karachi':19651127,}
population = pd.Series(population_dict)
print(population)
```

```
Lahore       38332521
Islamabad    26448193
Karachi      19651127
dtype: int64
```

2.3 DataFrame

A DataFrame represents a tabular, spreadsheet-like data structure containing an ordered collection of columns, each of which can be a different value type (numeric, string, Boolean, etc.). The DataFrame has both a row and column index; it can be thought of as a dict of Series. Like the Series object discussed in the previous section, the DataFrame can be thought of either as a generalization of a NumPy array, or as a specialization of a Python dictionary.

2.3.1 DataFrame as a generalized NumPy array

There are numerous ways to construct a DataFrame, though one of the most common is from a dict of equal-length lists or NumPy arrays

```
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'],
'year': [2000, 2001, 2002, 2001, 2002],
'population': [1.5, 1.7, 3.6, 2.4, 2.9]}
frame = pd.DataFrame(data)
print(frame)
```

```
    state  year  population
0    Ohio  2000         1.5
1    Ohio  2001         1.7
2    Ohio  2002         3.6
3  Nevada  2001         2.4
4  Nevada  2002         2.9
```

Another way using Series. Let's use the last example of series **population** we used in last section.

```
population_dict = {'Lahore': 38332521,'Islamabad': 26448193,'Karachi':19651127,}
population = pd.Series(population_dict)
area_dict = {'Lahore': 423967, 'Islamabad': 695662, 'Karachi': 141297}
area = pd.Series(area_dict)
states = pd.DataFrame({'population': population,'area': area})
print(states)
```

Output:

```
            population    area
Lahore        38332521  423967
Islamabad     26448193  695662
Karachi       19651127  141297
```

- Like the Series object, the DataFrame has an index attribute that gives access to the index labels
  ```
  print(states.index)
  Output:Index(['Lahore', 'Islamabad', 'Karachi'], dtype='object')
  ```

- Additionally, the DataFrame has a columns attribute, which is an Index object holding the column labels:
  ```
  print(states.columns)
  Output: Index(['population', 'area'], dtype='object')
  ```

- if you pass a column that isn't contained in data, it will appear with NA values in the result:

```
frame2 = pd.DataFrame(data, columns=['year', 'state', 'pop',
'debt'], index=['one', 'two', 'three', 'four', 'five'])
```

2.3.2 DataFrame as specialized dictionary

we can also think of a DataFrame as a specialization of a dictionary. Where a dictionary maps a key to a value, a DataFrame maps a column name to a Series of column data. For example, asking for the 'area' attribute returns the Series object containing the areas we saw earlier: states[‘area’]

Note: Notice the potential point of confusion here: in a two-dimensional NumPy array, data[0] will return the first row. For a DataFrame, data['col0'] will return the first column. Because of this, it is probably better to think about DataFrame as generalized dictionaries rather than generalized arrays, though both ways of looking at the situation can be useful.

2.4 Constructing DataFrame objects

A Pandas DataFrame can be constructed in a variety of ways

I. From a single Series object

- A DataFrame is a collection of Series objects, and a single column DataFrame can be constructed from a single Series.

```
oneDF = pd.DataFrame(population, columns=['population'])
```

II. From a list of dicts

- Any list of dictionaries can be made into a DataFrame. We’ll use a simple list comprehension to create some data

```
data = [{'a': i, 'b': 2 * i}
for i in range(3)]
print(pd.DataFrame(data))
```

Output:

```
   a  b
0  0  0
1  1  2
2  2  4
```

III. From a dictionary of Series objects

a DataFrame can be constructed from a dictionary of Series objects as well

```
population_dict = {'Lahore': 38332521,'Islamabad': 26448193,'Karachi':19651127,}
population = pd.Series(population_dict)
area_dict = {'Lahore': 423967, 'Islamabad': 695662, 'Karachi': 141297}
area = pd.Series(area_dict)
states = pd.DataFrame({'population': population,'area': area})
print(states)
```

IV. From a two-dimensional NumPy array

Given a two-dimensional array of data, we can create a DataFrame with any specified column and index names

```
data = pd.DataFrame(np.random.rand(3, 2),columns=['foo', 'bar'],index=['a', 'b', 'c'])
print(data)
```

Output:

```
        foo       bar
a  0.698446  0.075305
b  0.726363  0.935652
c  0.125593  0.124414
```

2.5 Reading Data files

Being able to create a DataFrame or Series by hand is handy. But, most of the time, we won't actually be creating our own data by hand. Instead, we'll be working with data that already exists.

Data can be stored in any of a number of different forms and formats. By far the most basic of these is the humble CSV file. CSV file is a table of values separated by commas. Hence the name: "Comma-Separated Values", or CSV.

```
data = pd.read_csv("\path/covid19.csv")
```

We can use the `shape` attribute to check how large the resulting DataFrame is:

```
print(data.shape)
```

We can examine the contents of the resultant DataFrame using the head() command, which grabs the first five rows:

print(data.head())

Note: The `pd.read_csv()` function is well-endowed, with over 30 optional parameters you can specify.

3. Access in Data Frame

In Python, we can access the property of an object by accessing it as an attribute. A book object, for example, might have a title property, which we can access by calling book. Title. Columns in a pandas DataFrame work in much the same way.

```
print(df.age)
Moreover, we can also access data frame as: print(df['age'][2])
```

4. Indexing in pandas

The indexing operator and attribute selection are nice because they work just like they do in the rest of the Python ecosystem. As a novice, this makes them easy to pick up and use. However, pandas has its own accessor operators, `loc` and `iloc`. For more advanced operations, these are the ones you're supposed to be using:

4.1 Index-based selection

Pandas indexing works in one of two paradigms. The first is index-based selection: selecting data based on its numerical position in the data. iloc follows this paradigm.

To select the first row of data in a DataFrame, we may use the following:

```
print(df.iloc[0])
```

```
age                                30
sex                              male
city          Chaohu City, Hefei City
province                       Anhui
country                        China
wuhan(0)_not_wuhan(1)              1
latitude                      31.647
longitude                    117.717
geo_resolution                admin3
date_onset_symptoms       18.01.2020
date_admission_hospital   20.01.2020
date_confirmation         22.01.2020
symptoms                         NaN
lives_in_Wuhan                   yes
travel_history_dates      17.01.2020
travel_history_location        Wuhan
```

Both loc and iloc are row-first, column-second. This is the opposite of what we do in native Python, which is column-first, row-second.

This means that it's marginally easier to retrieve rows, and marginally harder to get retrieve columns. To get a column with iloc, we can do the following: print(df.iloc[:,0])

to select the column from just the first, second, and third row, we would do: print(df.iloc[:3,0])

```
ID
1.0    30
2.0    47
3.0    49
Name: age, dtype: object
```

It's also possible to pass a list: `print(df.iloc[[0,1,2],0])`

Finally, it's worth knowing that negative numbers can be used in selection. This will start counting forwards from the *end* of the values. So, for example here are the last five elements of the dataset.

```
print(df.iloc[-5:])
```

4.2 Label-based selection

The second paradigm for attribute selection is the one followed by the loc operator: label-based selection. In this paradigm, it's the data index value, not its position, which matters.

For example, to get the first entry in covid19, we would now do the following:
print(df.loc[1,'age'])

iloc is conceptually simpler than loc because it ignores the dataset's indices. When we use iloc we treat the dataset like a big matrix (a list of lists), one that we have to index into by position. loc, by contrast, uses the information in the indices to do its work. Since your dataset usually has meaningful indices, it's usually easier to do things using loc instead. For example, here's one operation that's much easier using loc:

```
print(df.loc[:,['age','city','province']])
```

Note: `iloc` uses the Python stdlib indexing scheme, where the first element of the range is included and the last one excluded. So `0:10` will select entries `0,...,9`. `loc`, meanwhile, indexes inclusively. So `0:10` will select entries `0,...,10`.

5. Conditional Selection

So far we've been indexing various strides of data, using structural properties of the DataFrame itself. To do *interesting* things with the data, however, we often need to ask questions based on conditions.

For example, suppose that we're interested specifically in Anhui province cases

print(df.loc[(df.province == 'Anhui')]) total 788 records found

We can use the or (|) to bring the two questions together:

```
print(df.loc[(df.province == 'Anhui') | (df.age == 30)])
```

Pandas comes with a few built-in conditional selectors, two of which we will highlight here.

The first is isin. isin is lets you select data whose value "is in" a list of values. For example case in in Anhui or Beijing.

```
print(df.loc[(df.province.isin(['Anhui','Beijing']))])
```

The second is `isnull` (and its companion `notnull`). These methods let you highlight values which are (or are not) empty (`NaN`). For example, to filter out wines lacking a price tag in the dataset, here's what we would do:

```
print(df.loc[(df.age.notnull())])
```

Assigning data:

Going the other way, assigning data to a DataFrame is easy. You can assign either a constant value:

```
df['new'] = 'test-case'
print(df['new'])
```

6. Functions and maps

To see a list of unique values we can use the `unique()` function:
df.age.unique()

```
print(data.Member_number.unique())
```

More functions which are available with pandas

	Function	Usage	Comments
1	pd.read_csv()	Read CSV file	.csv, .tsv or .txt
2	pd.read_excel()	Read spreadsheet	.xls or .xlsx
3	pd.read_json()	Read JSON document	.json
4	pd.read_sql()	Read directly from DB query	Needs 3rd party library to support (e.g. s
5	df.to_csv()	Write to CSV file	.csv, .tsv or .txt
6	df.to_excel()	Write to Excel file	.xls or .xlsx
7	df.to_json()	Write to JSON document	.json
8	df.to_sql()	Write to DB table	Needs 3rd party library to support (e.g. s

Name	Usage	Comments
df.head()	Preview the first n (default=5)	Can define "n" by df.head(n
df.tail()	Preview the last n (default=5)	Can define "n" by df.tail(n)
df.sort_values()	Sort the data frame on a specific column	Can sort with multiple colur
df.columns	Display all column names	Can also set column names
df.dtypes	Display data types of the columns	Return a list of types
df.shape	Display the shape of the data frame	Return a tuple: (row_count,
df.describe()	Show basic stats of each column	Will show different stats for
s.value_counts()	Count occurrences of each value	Use df['...'] to get a column

Name	Usage	Comments
s.isna()	Whether there are null values existing	Boolean
df.dropna()	Delete missing values	Can be applied on a series or
df.fillna()	Fill missing values with a certain value	Can be applied on a series or
df.drop_duplicates()	Delete all duplicated values	Can be applied on a series or
df.drop()	Delete columns or rows	Need to specify axis
df.rename()	Rename columns or rows	Need to specify axis
df.reset_index()	Convert index column to a data column	A new index will be automatic

Name	Usage
pd.concat()	Concatenate two dataframe
pd.merge()	Joining two data frames
df.groupby()	Grouping column(s) and then apply more aggregating functions
df.groupby().agg()	Aggregating records for each group
pd.pivot_table()	Convert data frame to pivot table

Important:

```
# Read from local path
df1 = pd.read_csv('./data.csv')
df2 = pd.read_excel('./data.xlsx', sheet_name='Sheet1')
df3 = pd.read_json('http://example.com/data.json', orient='records')
```

Writing to files

```
df1.to_csv('./data.csv')
df2.to_excel('./data.xlsx')
df3.to_json('./data.json')
```

`df.describe()` | Summary statistics for numerical columns
`df.mean()` | Returns the mean of all columns
`df.corr()` | Returns the correlation between columns in a DataFrame
`df.count()` | Returns the number of non-null values in each DataFrame column
`df.max()` | Returns the highest value in each column
`df.min()` | Returns the lowest value in each column
`df.median()` | Returns the median of each column
`df.std()` | Returns the standard deviation of each column

Combining:

When performing operations on a dataset, we will sometimes need to combine different DataFrames and/or Series in non-trivial ways. Pandas has three core methods for doing this. In order of increasing complexity, these are `concat()`, `join()`, and `merge()`.

```
data = pd.read_csv("path/Groceries.csv")
data1 = pd.read_csv("path/covid19.csv")
print(pd.concat([data, data1]))
```

7. Evaluation Task (Unseen) [Expected time = 30mins for tasks]

I will give you unseen task depending upon the progress of the class.

7.1 Practice Task 1

Create a DataFrame fruits that looks like this:

	Bananas	Apples
0	20	17

7.2 Practice Task 2

Create a DataFrame sales that looks like below:

	Bananas	Apples
September	200	120
October	165	90

7.3 Practice Task 3

Create a series variable marks that looks like:

```
MIS      67
DBMS     93
A&F      56
CA       67
dtype: int64
```

7.4 Practice Task 4

Read the following csv dataset of Groceries given to you in zip folder into a DataFrame called grocer

7.5 Practice Task 5

Select the itemdescription column from grocer and assign the result to the variable itemdesc.

7.6 Practice Task 6

Select the first value from the member column of grocer and assign it to variable first_member.

7.7 Practice Task 7

Select the first 20 row of data and save it to variable first_rows.

7.8 Practice Task 8

Create a variable selected_grocer containing the values of date and item description column only for purchase made in 2015

7.9 Practice Task 9

Create a variable containing the member and date columns of the first 500 records.

7.10 Practice Task 10

Create a DataFrame member haivng purchases both in 2014 and 2015 year. Hint: grocer.member and grocer.date check??

8. Evaluation criteria

The evaluation criteria for this lab will be based on the completion of the following tasks. Each task is assigned the marks percentage which will be evaluated by the instructor in the lab whether the student has finished the complete/partial task(s).

Table 3: Evaluation of the Labs

Sr. No.	Description	Marks
1	Problem Modeling	20
2	Procedures and Tools	10
3	Practice tasks and Testing	35
4	Evaluation Tasks (Unseen)	20
5	Comments	5
6	Good Programming Practices	10