Object Oriented Programming (Using Python)

UNIT- III

Python Libraries:

- Pandas
  - Merge
  - Grouping
  - Time series
  - Categoricals
  - Importing and exporting data

Prof. R. MADANA MOHANA
Professor, Artificial Intelligence & Data Science
http://rmadanamohana.com/

Quick overview of pandas?

- https://pandas.pydata.org/docs/user_guide/10min.html#min

Merge

Concat

pandas provides various facilities for easily combining together Series and DataFrame objects with various kinds of set logic for the indexes and relational algebra functionality in the case of join / merge-type operations.

Merge

Concat

Concatenating pandas objects together along an axis with concat():

```
>>> df = pd.DataFrame(np.random.randn(10, 4))
>>> df
          0         1         2         3
0 -0.717196  0.473925 -0.395855 -1.621849
1 -0.219668 -1.228870 -0.914974 -0.241247
2  2.244973 -0.415200  0.056994  0.101080
3 -0.878046  0.416592  1.453482 -0.033171
4  1.114444 -1.857564 -0.410036 -0.823989
5 -1.025485  0.487437 -0.406924  1.596674
6 -0.001355  1.356347  0.042176  0.712419
7  1.518298 -0.282735  0.731253 -1.149422
8  0.862762 -0.563118  0.055865  0.801844
9 -0.608196 -1.381167  0.419710  0.268061
```

Merge

Concat

Concatenating pandas objects together along an axis with concat():

```
>>> pieces = [df[:3], df[3:7], df[7:]]
>>> pieces
[          0         1         2         3
0 -0.717196  0.473925 -0.395855 -1.621849
1 -0.219668 -1.228870 -0.914974 -0.241247
2  2.244973 -0.415200  0.056994  0.101080,
          0         1         2         3
3 -0.878046  0.416592  1.453482 -0.033171
4  1.114444 -1.857564 -0.410036 -0.823989
5 -1.025485  0.487437 -0.406924  1.596674
6 -0.001355  1.356347  0.042176  0.712419,
          0         1         2         3
7  1.518298 -0.282735  0.731253 -1.149422
8  0.862762 -0.563118  0.055865  0.801844
9 -0.608196 -1.381167  0.419710  0.268061]
```

Merge

Concat

Concatenating pandas objects together along an axis with concat():

```
>>> pd.concat(pieces)
          0         1         2         3
0 -0.717196  0.473925 -0.395855 -1.621849
1 -0.219668 -1.228870 -0.914974 -0.241247
2  2.244973 -0.415200  0.056994  0.101080
3 -0.878046  0.416592  1.453482 -0.033171
4  1.114444 -1.857564 -0.410036 -0.823989
5 -1.025485  0.487437 -0.406924  1.596674
6 -0.001355  1.356347  0.042176  0.712419
7  1.518298 -0.282735  0.731253 -1.149422
8  0.862762 -0.563118  0.055865  0.801844
9 -0.608196 -1.381167  0.419710  0.268061
```

Merge

Join

merge() enables SQL style join types along specific columns:

```
>>> import numpy as np
>>> import pandas as pd
>>> left = pd.DataFrame({"key": ["oop", "oop"], "lval": [1, 2]})
>>> left
   key  lval
0  oop     1
1  oop     2
```

Merge

Join

```
>>> right = pd.DataFrame({"key": ["oop", "oop"], "rval": [4, 5]})
>>> right
   key  rval
0  oop     4
1  oop     5
>>> pd.merge(left, right, on="key")
   key  lval  rval
0  oop     1     4
1  oop     1     5
2  oop     2     4
3  oop     2     5
```

Merge

Join – Example2

```
>>> left = pd.DataFrame({"key": ["oop", "python"], "lval": [1, 2]})
>>> left
      key  lval
0     oop     1
1  python     2
>>>   right   =   pd.DataFrame({"key":   ["oop",   "python"], "rval": [4, 5]})
>>> right
      key  rval
0     oop     4
1  python     5
```

Merge

Join – Example2

```
>>> pd.merge(left, right, on="key")
      key  lval  rval
0     oop     1     4
1  python     2     5
```

Grouping

By “group by” we are referring to a process involving one or more of the following steps:

- **Splitting** the data into groups based on some criteria
- **Applying** a function to each group independently
- **Combining** the results into a data structure

Grouping

```
>>> import numpy as np
>>> import pandas as pd
>>> df = pd.DataFrame(
    {
        "A": ["oop", "python", "oop", "python", "oop", "python", "oop", "oop"],
        "B": ["one", "one", "two", "three", "two", "two", "one", "three"],
        "C": np.random.randn(8),
        "D": np.random.randn(8),
    }
)
```

Grouping

```
>>> df
      A       B         C         D
0   oop     one  0.507365  1.299938
1  python   one -0.829337 -0.672491
2   oop     two -1.725394 -1.049454
3  python three  0.739141  1.984817
4   oop     two  1.058922 -1.549976
5  python   two -0.285935 -0.435408
6   oop     one -1.351595  0.362258
7   oop   three -0.005763  1.304180
```

Grouping

Grouping and then applying the `sum()` function to the resulting groups:

```
>>> df
      A       B         C         D
0     oop     one  0.507365  1.299938
1  python     one -0.829337 -0.672491
2     oop     two -1.725394 -1.049454
3  python   three  0.739141  1.984817
4     oop     two  1.058922 -1.549976
5  python     two -0.285935 -0.435408
6     oop     one -1.351595  0.362258
7     oop   three -0.005763  1.304180
```

Grouping

Grouping and then applying the `sum()` function to the resulting groups:

```
>>> df.groupby("A")[["C", "D"]].sum()
                C         D
A
oop    -1.516466  0.366946
python -0.376131  0.876918
```

Grouping

Grouping by multiple columns forms a hierarchical index, and again we can apply the `sum()` function:

```
>>> df.groupby(["A", "B"]).sum()
                  C         D
A      B
oop    one   -0.844231  1.662196
       three -0.005763  1.304180
       two   -0.666472 -2.599430
python one   -0.829337 -0.672491
       three  0.739141  1.984817
       two   -0.285935 -0.435408
```

Time series

- pandas has simple, powerful, and efficient functionality for performing resampling operations during frequency conversion (e.g., converting secondly data into 5-minutely data).
- This is extremely common in, but not limited to, financial applications.

```
>>> rng = pd.date_range("19/06/2023", periods=100, freq="S")
>>> rng
```

Time series

IDLE Shell 3.10.0

File Edit Shell Debug Options Window Help

```
>>> rng
DatetimeIndex(['2023-06-19 00:00:00', '2023-06-19 00:00:01',
               '2023-06-19 00:00:02', '2023-06-19 00:00:03',
               '2023-06-19 00:00:04', '2023-06-19 00:00:05',
               '2023-06-19 00:00:06', '2023-06-19 00:00:07',
               '2023-06-19 00:00:08', '2023-06-19 00:00:09',
               '2023-06-19 00:00:10', '2023-06-19 00:00:11',
               '2023-06-19 00:00:12', '2023-06-19 00:00:13',
               '2023-06-19 00:00:14', '2023-06-19 00:00:15',
               '2023-06-19 00:00:16', '2023-06-19 00:00:17',
               '2023-06-19 00:00:18', '2023-06-19 00:00:19',
               '2023-06-19 00:00:20', '2023-06-19 00:00:21',
               '2023-06-19 00:00:22', '2023-06-19 00:00:23',
               '2023-06-19 00:00:24', '2023-06-19 00:00:25',
               '2023-06-19 00:00:26', '2023-06-19 00:00:27',
               '2023-06-19 00:00:28', '2023-06-19 00:00:29',
               '2023-06-19 00:00:30', '2023-06-19 00:00:31',
               '2023-06-19 00:00:32', '2023-06-19 00:00:33',
               '2023-06-19 00:00:34', '2023-06-19 00:00:35',
               '2023-06-19 00:00:36', '2023-06-19 00:00:37',
               '2023-06-19 00:00:38', '2023-06-19 00:00:39',
               '2023-06-19 00:00:40', '2023-06-19 00:00:41',
               '2023-06-19 00:00:42', '2023-06-19 00:00:43',
               '2023-06-19 00:00:44', '2023-06-19 00:00:45',
               '2023-06-19 00:00:46', '2023-06-19 00:00:47',
```

IDLE Shell 3.10.0

File Edit Shell Debug Options Window Help

```
               '2023-06-19 00:00:52', '2023-06-19 00:00:53',
               '2023-06-19 00:00:54', '2023-06-19 00:00:55',
               '2023-06-19 00:00:56', '2023-06-19 00:00:57',
               '2023-06-19 00:00:58', '2023-06-19 00:00:59',
               '2023-06-19 00:01:00', '2023-06-19 00:01:01',
               '2023-06-19 00:01:02', '2023-06-19 00:01:03',
               '2023-06-19 00:01:04', '2023-06-19 00:01:05',
               '2023-06-19 00:01:06', '2023-06-19 00:01:07',
               '2023-06-19 00:01:08', '2023-06-19 00:01:09',
               '2023-06-19 00:01:10', '2023-06-19 00:01:11',
               '2023-06-19 00:01:12', '2023-06-19 00:01:13',
               '2023-06-19 00:01:14', '2023-06-19 00:01:15',
               '2023-06-19 00:01:16', '2023-06-19 00:01:17',
               '2023-06-19 00:01:18', '2023-06-19 00:01:19',
               '2023-06-19 00:01:20', '2023-06-19 00:01:21',
               '2023-06-19 00:01:22', '2023-06-19 00:01:23',
               '2023-06-19 00:01:24', '2023-06-19 00:01:25',
               '2023-06-19 00:01:26', '2023-06-19 00:01:27',
               '2023-06-19 00:01:28', '2023-06-19 00:01:29',
               '2023-06-19 00:01:30', '2023-06-19 00:01:31',
               '2023-06-19 00:01:32', '2023-06-19 00:01:33',
               '2023-06-19 00:01:34', '2023-06-19 00:01:35',
               '2023-06-19 00:01:36', '2023-06-19 00:01:37',
               '2023-06-19 00:01:38', '2023-06-19 00:01:39'],
              dtype='datetime64[ns]', freq='S')
```

Time series

```
>>> ts = pd.Series(np.random.randint(0, 500, len(rng)), index=rng)
>>> ts
2023-06-19 00:00:00    285
2023-06-19 00:00:01    414
2023-06-19 00:00:02    357
2023-06-19 00:00:03    328
2023-06-19 00:00:04     52
                      ...
2023-06-19 00:01:35    499
2023-06-19 00:01:36    482
2023-06-19 00:01:37    396
2023-06-19 00:01:38    259
2023-06-19 00:01:39     27
Freq: S, Length: 100, dtype: int32
```

Time series

```
>>> ts.resample("5Min").sum()
2023-06-19    24777
Freq: 5T, dtype: int32
```

Time series

- Series.tz_localize() localizes a time series to a time zone:

```
>>> rng = pd.date_range("19/06/2023 00:00", periods=5, freq="D")
>>> rng
DatetimeIndex(['2023-06-19', '2023-06-20', '2023-06-21', '2023-06-22', '2023-06-23'],
              dtype='datetime64[ns]', freq='D')
```

Time series

- Series.tz_localize() localizes a time series to a time zone:

```
>>> ts = pd.Series(np.random.randn(len(rng)), rng)
>>> ts
2023-06-19   -0.404256
2023-06-20    0.477535
2023-06-21    2.293098
2023-06-22   -0.408688
2023-06-23    0.972186
Freq: D, dtype: float64
```

Time series

- Series.tz_localize() localizes a time series to a time zone:

```
>>> ts_utc = ts.tz_localize("UTC")
>>> ts_utc
2023-06-19 00:00:00+00:00   -0.404256
2023-06-20 00:00:00+00:00    0.477535
2023-06-21 00:00:00+00:00    2.293098
2023-06-22 00:00:00+00:00   -0.408688
2023-06-23 00:00:00+00:00    0.972186
Freq: D, dtype: float64
```

Time series

- Series.tz_convert() converts a timezones aware time series to another time zone:
- aware object represents a specific moment in time that is not open to interpretation.

```
>>> ts_utc = ts.tz_localize("UTC")
>>> ts_utc
2023-06-19 00:00:00+00:00   -0.404256
2023-06-20 00:00:00+00:00    0.477535
2023-06-21 00:00:00+00:00    2.293098
2023-06-22 00:00:00+00:00   -0.408688
2023-06-23 00:00:00+00:00    0.972186
Freq: D, dtype: float64
```

Time series

- Series.tz_convert() converts a timezones aware time series to another time zone:

```
>>> ts_utc.tz_convert("US/Eastern")
2023-06-18 20:00:00-04:00   -0.404256
2023-06-19 20:00:00-04:00    0.477535
2023-06-20 20:00:00-04:00    2.293098
2023-06-21 20:00:00-04:00   -0.408688
2023-06-22 20:00:00-04:00    0.972186
Freq: D, dtype: float64
```

Time series

- Converting between time span representations:

```
>>> rng = pd.date_range("1/1/2023", periods=5, freq="M")
>>> rng
DatetimeIndex(['2023-01-31', '2023-02-28', '2023-03-31',
'2023-04-30',
               '2023-05-31'],
              dtype='datetime64[ns]', freq='M')
```

Time series

- Converting between time span representations:

```
>>> ts = pd.Series(np.random.randn(len(rng)), index=rng)
>>> ts
2023-01-31   -0.457527
2023-02-28   -0.320943
2023-03-31   -0.910579
2023-04-30   -1.829910
2023-05-31    1.248885
Freq: M, dtype: float64
```

Time series

- Converting between time span representations:

```
>>> ps = ts.to_period()
>>> ps
2023-01   -0.457527
2023-02   -0.320943
2023-03   -0.910579
2023-04   -1.829910
2023-05    1.248885
Freq: M, dtype: float64
```

Time series

- Converting between time span representations:

```
>>> ps.to_timestamp()
2023-01-01   -0.457527
2023-02-01   -0.320943
2023-03-01   -0.910579
2023-04-01   -1.829910
2023-05-01    1.248885
Freq: MS, dtype: float64
```

Categoricals

- pandas can include categorical data in a DataFrame.
- Categoricals are a pandas data type corresponding to categorical variables in statistics.
- A categorical variable takes on a limited, and usually fixed, number of possible values.
- Examples are gender, social class, blood type, country affiliation, observation time or rating via Likert scales.
- In contrast to statistical categorical variables, categorical data might have an order (e.g. ‘strongly agree’ vs ‘agree’ or ‘first observation’ vs. ‘second observation’), but numerical operations (additions, divisions, …) are not possible.
- All values of categorical data are either in `categories` or `np.nan`. Order is defined by the order of `categories`, not lexical order of the values. Internally, the data structure consists of a `categories` array and an integer array of `codes` which point to the real value in the `categories` array.

Categoricals

```
>>> df = pd.DataFrame(
    {"id": [1, 2, 3, 4, 5, 6], "raw_grade": ["a", "b", "b", "a",
"a", "e"]}
)
>>> df
   id raw_grade
0   1         a
1   2         b
2   3         b
3   4         a
4   5         a
5   6         e
```

Categoricals

Converting the raw grades to a categorical data type:

```
>>> df["grade"] = df["raw_grade"].astype("category")
>>> df["grade"]
0    a
1    b
2    b
3    a
4    a
5    e
Name: grade, dtype: category
Categories (3, object): ['a', 'b', 'e']
```

Categoricals

Rename the categories to more meaningful names:

```
>>> new_categories = ["very good", "good", "very bad"]
>>> df["grade"]= df["grade"].cat.rename_categories(new_categories)
>>> df["grade"]
0     very good
1          good
2          good
3     very good
4     very good
5      very bad
Name: grade, dtype: category
Categories (3, object): ['very good', 'good', 'very bad']
```

Categoricals

Reorder the categories and simultaneously add the missing categories (methods under `Series.cat()` return a new `Series` by default):

```
>>> df["grade"] = df["grade"].cat.set_categories(
    ["very bad", "bad", "medium", "good", "very good"]
)
>>> df["grade"]
0    very good
1         good
2         good
3    very good
4    very good
5     very bad
Name: grade, dtype: category
Categories (5, object): ['very bad', 'bad', 'medium', 'good', 'very good']
```

Categoricals

Sorting is per order in the categories, not lexical order:

```
>>> df.sort_values(by="grade")
   id  raw_grade      grade
5   6          e   very bad
1   2          b       good
2   3          b       good
0   1          a  very good
3   4          a  very good
4   5          a  very good
```

Categoricals

Grouping by a categorical column also shows empty categories:

```
>>> df.groupby("grade").size()
grade
very bad      1
bad           0
medium        0
good          2
very good     3
dtype: int64
```

Importing and exporting data

CSV

Reading from a csv file: using `read_csv()`:

```
"C:/Users/rmmna/Downloads/food.csv"
```

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	Category	Description	Nutrient D	Data.Alph	Data.Beta	Data.Beta	Data.Carb	Data.Chol	Data.Chol	Data.Fiber	Data.Lutei	Data.Lyco	Data.Niaci	Data.Prote	Data.Retir
2	Milk	Milk, human	11000000	0	7	0	6.89	14	16	0	0	0	0.177	1.03	60
3	Milk	Milk, NFS	11100000	0	4	0	4.87	8	17.9	0	0	0	0.11	3.34	58
4	Milk	Milk, whole	11111000	0	7	0	4.67	12	17.8	0	0	0	0.105	3.28	31
5	Milk	Milk, low sodium, whole	11111100	0	7	0	4.46	14	16	0	0	0	0.043	3.1	28
6	Milk	Milk, calcium fortified, whole	11111150	0	7	0	4.67	12	17.8	0	0	0	0.105	3.28	31
7	Milk	Milk, calcium fortified, low fat (1%)	11111160	0	1	0	5.19	5	17.4	0	0	0	0.113	3.38	58
8	Milk	Milk, calcium fortified, fat free (skim)	11111170	0	0	0	4.85	2	16	0	0	0	0.088	3.4	137
9	Milk	Milk, reduced fat (2%)	11112110	0	3	0	4.91	8	18.2	0	0	0	0.112	3.35	83
10	Milk	Milk, acidophilus, low fat (1%)	11112120	0	1	0	5.19	5	17.4	0	0	0	0.113	3.38	58
11	Milk	Milk, acidophilus, reduced fat (2%)	11112130	0	3	0	4.91	8	18.2	0	0	0	0.112	3.35	83
12	Milk	Milk, low fat (1%)	11112210	0	1	0	5.19	5	17.4	0	0	0	0.113	3.38	58
13	Milk	Milk, fat free (skim)	11113000	0	2	0	4.89	3	18.2	0	0	0	0.118	3.43	64
14	Milk	Milk, lactose free, low fat (1%)	11114300	0	1	0	5.19	5	17.4	0	0	0	0.113	3.38	58
15	Milk	Milk, lactose free, fat free (skim)	11114320	0	2	0	4.89	3	18.2	0	0	0	0.118	3.43	64
16	Milk	Milk, lactose free, reduced fat (2%)	11114330	0	3	0	4.91	8	18.2	0	0	0	0.112	3.35	83
17	Milk	Milk, lactose free, whole	11114350	0	7	0	4.67	12	17.8	0	0	0	0.105	3.28	31
18	Buttermilk	Buttermilk, fat free (skim)	11115000	0	1	0	4.79	4	17.7	0	0	0	0.058	3.31	13
19	Buttermilk	Buttermilk, low fat (1%)	11115100	0	1	0	4.79	4	17.7	0	0	0	0.058	3.31	13
20	Buttermilk	Buttermilk, reduced fat (2%)	11115200	0	3	0	5.3	8	16	0	0	0	0.1	4.1	16
21	Buttermilk	Buttermilk, whole	11115300	0	7	0	4.88	11	14.6	0	0	0	0.09	3.21	46
22	Kefir	Kefir, NS as to fat content	11115400	0	2	0	7.48	5	15.2	0	0	0	0.128	3.59	174
23	Goat's mil	Goat's milk, whole	11116000	0	7	0	4.45	11	16	0	0	0	0.277	3.56	56

food

Importing and exporting data

CSV

Reading from a csv file: using read_csv():

```
>>> pd.read_csv("C:/Users/rmmna/Downloads/food.csv")
```

```
                                     Category  ... Data.Vitamins.Vitamin K
0                                        Milk  ...                    0.3
1                                        Milk  ...                    0.2
2                                        Milk  ...                    0.3
3                                        Milk  ...                    0.3
4                                        Milk  ...                    0.3
...                                       ...  ...                    ...
7078          Tomatoes as ingredient in omelet  ...                    8.8
7079  Other vegetables as ingredient in omelet  ...                    0.4
7080         Vegetables as ingredient in curry  ...                    8.9
7081         Sauce as ingredient in hamburgers  ...                   50.8
7082      Industrial oil as ingredient in food  ...                  155.8
[7083 rows x 38 columns]
```

Importing and exporting data

CSV

Writing to a csv file: using `DataFrame.to_csv()`:

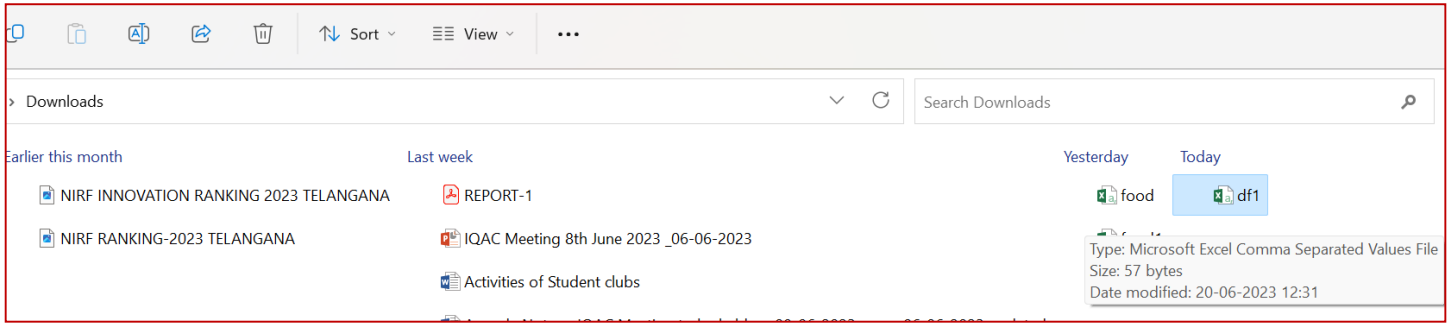
```
>>> import pandas as pd
>>> df = pd.DataFrame(
    {"id": [1, 2, 3, 4, 5, 6], "raw_grade": ["a", "b", "b", "a", "a", "e"]}
)
>>> df
   id    raw_grade
0   1          a
1   2          b
2   3          b
3   4          a
4   5          a
5   6          e
```

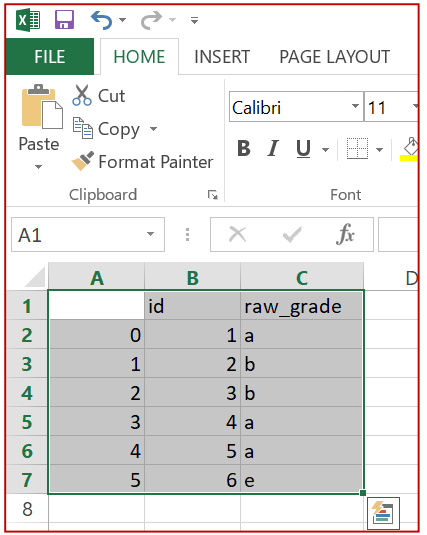
Importing and exporting data

CSV

Writing to a csv file: using `DataFrame.to_csv()`:

```
>>> df.to_csv("C:/Users/rmmna/Downloads/df1.csv")
```





Importing and exporting data

Excel

Writing to an excel file using `DataFrame.to_excel()`:

```
>>> import pandas as pd
>>> df1 = pd.DataFrame([['a', 'b'], ['c', 'd']],
                       index=['row 1', 'row 2'],
                       columns=['col 1', 'col 2'])
>>> df1
```

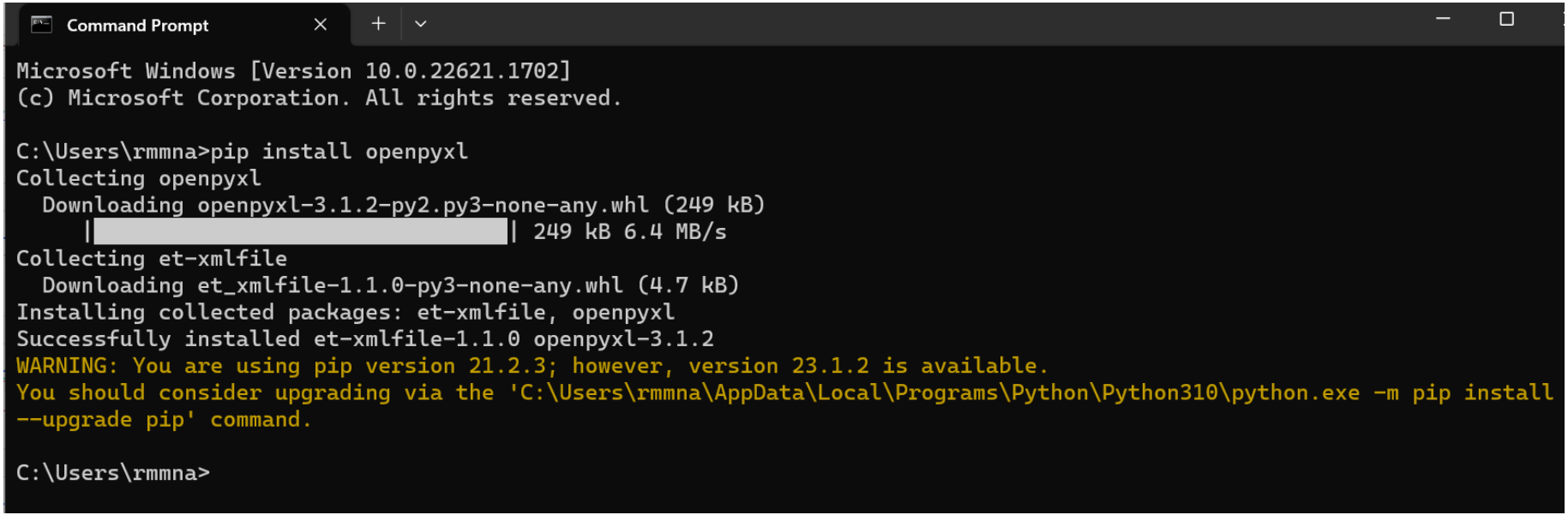
```
      col 1 col 2
row 1     a     b
row 2     c     d
```

Importing and exporting data

Excel

Writing to an excel file using `DataFrame.to_excel()`:

- install the module by running the `pip install openpyxl` command.

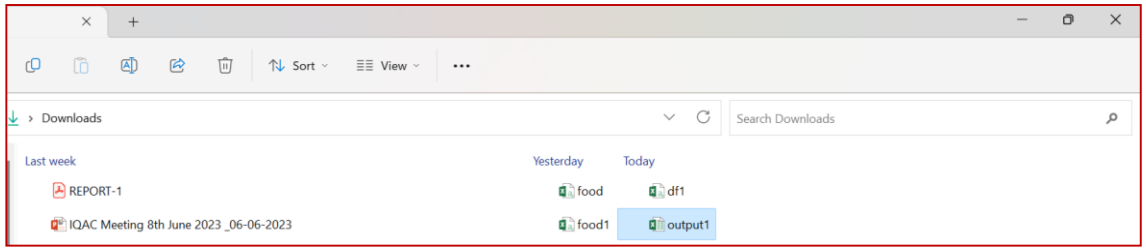


Importing and exporting data

Excel

Writing to an excel file using `DataFrame.to_excel()`:

```
>>> df1.to_excel("C:/Users/rmmna/Downloads/output1.xlsx")
```



	A	B	C
1		col 1	col 2
2	row 1	a	b
3	row 2	c	d
4			

Importing and exporting data

Excel

Reading from an excel file using `read_excel()`:

```
>>>pd.read_excel("C:/Users/rmmna/Downloads/output1.xlsx",
"Sheet1", index_col=None, na_values=["NA"])
```

```
 Unnamed: 0 col 1 col 2
0       row 1     a     b
1       row 2     c     d
```

Lab Experiment

Write a Pandas program to convert a NumPy array to a Pandas series

```
>>> import numpy as np
>>> import pandas as pd
# Define a NumPy array
>>> data = np.array([1, 2, 3, 4, 5])
>>> data
array([1, 2, 3, 4, 5])
>>> print(data)
[1 2 3 4 5]
```

Lab Experiment

Write a Pandas program to convert a NumPy array to a Pandas series cont'd.

```
# Convert the array to a Pandas Series
>>> data_series = pd.Series(data)
>>> data_series
0    1
1    2
2    3
3    4
4    5
dtype: int32
>>> print(data_series)
```

Lab Experiment

Write a Pandas program to convert a NumPy array to a Pandas series cont'd.

In this example,

- we first import the NumPy and Pandas libraries.
- Then, we define a NumPy array '`data`' with some sample data.
- We then use the '`pd.Series()`' function to convert the array to a Pandas Series, and store the result in the '`data_series`' variable.
- Finally, we print the Series using the '`print()`' function.