ICHEC
Irish Centre for High-End Computing

PRACE

SUMMER OF HPC

BASIC DATA ANALYSIS WITH

pandas

Pandas – Data manipulation tool

- Pandas is a efficient tool for handling and manipulating “relational” or “labelled” data in Python in a easy and intuitive way.
  - Several file format are supported (‘.csv’, ‘.json’, ‘.txt’, ‘.xlsx’,…)
  - Good for both ordered and unordered time series data.
  - Great tool for observational and statistical data sets.
- Pandas is built upon two main objects:
  - DataFrame
  - Series

```
>>> import pandas as pd
```

Pandas Series

- Intuitively series are comparable to Python dictionaries but data processing and storing is more efficient.

- Creating a series:

```
>>> pd.Series(data, index=index)
```

  - From a list:

```
>>> series = pd.Series([1, 2, 3, 4],
... index=['a', 'b', 'c', 'd'])
... print(series)
a    1
b    2
c    3
d    4
dtype: int64
```

Pandas Series

- Intuitively series are comparable to Python dictionaries but data processing and storing is more efficient.

- Creating a series:

```
>>> pd.Series(data, index=index)
```

  - From a dictionary:

```
>>> dict_pop = {'Dublin':1110627,
...             'Belfast': 579726,
...             'Cork': 198582,
...             'Derry': 93512}
... series_pop = pd.Series(dict_pop)
... print(series_pop)
Dublin      1110627
Belfast      579726
Cork         198582
Derry         93512
dtype: int64
```

Pandas Series

- Intuitively series are comparable to Python dictionaries but data processing and storing is more efficient.

- Creating a series:

```
>>> pd.Series(data, index=index)
```

  - Series with special indexing:

```
>>> series = pd.Series([1, 2, 3, 4],
...          index=['a', 'b', 'c', 'd'])
... print(series)
a    1
b    2
c    3
d    4
dtype: int64
```

Pandas DataFrame

- Whereas **Series** are single columns, a DataFrame can be thought as a relational database, with several rows and named columns.

- A general syntax for creating a DataFrame:

```
>>> pd.Series(data, index=index)
```

  - An example creating a DataFrame from an existing Pandas series:

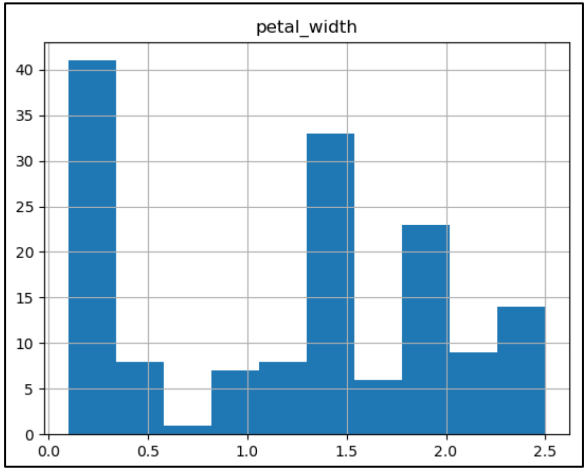
```
>>> MyDataFrame = pd.DataFrame(series_pop,columns=['POPULATION'])
... print(MyDataFrame)
          POPULATION
Dublin       1110627
Belfast       579726
Cork          198582
Derry          93512
```

Pandas DataFrame

- `DataFrame.head(n)` and `DataFrame.tail(n)` will display the n first and last rows of the DataFrame, respectively.

- We can easily graph our data. For example, if we want to plot the histogram distribution of a column:

```
>>> import matplotlib.pyplot as plt
... plt.show()
... hist = Iris.hist('petal_width')
```



Data: https://gist.github.com/curran/a08a1080b88344b0c8a7#file-iris-csv

Pandas DataFrame I/O

- More often, DataFrames are created from data files. We have several methods for different formats:
  - read_json()
  - read_html()
  - read_sql()
  - read_pickle()

```
>>> Iris = pd.read_csv('iris.csv')
>>> Iris.head()
   sepal_length  sepal_width  petal_length  petal_width species
0           5.1          3.5           1.4          0.2  setosa
1           4.9          3.0           1.4          0.2  setosa
2           4.7          3.2           1.3          0.2  setosa
3           4.6          3.1           1.5          0.2  setosa
4           5.0          3.6           1.4          0.2  setosa
```

Data: https://gist.github.com/curran/a08a1080b88344b0c8a7#file-iris-csv

Pandas DataFrame I/O

- Read methods offer many options to customize how pandas should read the file. We can choose the delimiter, desired columns, header…

```
>>> pandas.read_csv(filepath_or_buffer, delimiter=',',
...                 header='infer',
...                 names=0,
...                 index_col='index_col_name',
...                 usecols=['col1','col2'])
```

- Similarly, a DataFrame can be written into a file using:

```
>>> MyDataFrame.to_csv(r'path\filename.csv', index = False, header=True)
```

Accessing and manipulating data

- Let's consider the following DataFrame:

```
>>> n = 20
... PeopleDF = pd.DataFrame(dict(
...     AGE=np.random.randint(10,80, size=n),
...     HEIGHT=np.random.randint(150, 200, size=n)),
...     index = ['Person'+str(i+1) for i in range(n)])
... print(PeopleDF.head())
         AGE  HEIGHT
Person1   40     180
Person2   52     189
Person3   47     158
Person4   19     197
Person5   17     160
```

- Accessing a column
- Accessing a row
- Manipulating values
- Renaming columns
- Re-indexing

Accessing and manipulating data

- Accessing a column

Original DataFrame:

```
        AGE  HEIGHT
Person1  40     180
Person2  52     189
Person3  47     158
Person4  19     197
Person5  17     160
```

```
>>> PeopleDF.head()['AGE']
Person1    40
Person2    52
Person3    47
Person4    19
Person5    17
Name: AGE, dtype: int32
```

The result is a Pandas Series:

```
>>> type(PeopleDF.head()['AGE'])
<class 'pandas.core.series.Series'>
```

Accessing and manipulating data

- Accessing a row

Original DataFrame:

```
         AGE  HEIGHT
Person1   40     180
Person2   52     189
Person3   47     158
Person4   19     197
Person5   17     160
```

Two ways to do this:

```
>>> PeopleDF.loc['Person1']
AGE        40
HEIGHT    180
Name: Person1, dtype: int32
```

```
>>> PeopleDF.iloc[0]
AGE        40
HEIGHT    180
Name: Person1, dtype: int32
```

Accessing and manipulating data

- Accessing the elements of the DataFrame

Original DataFrame:

```
         AGE  HEIGHT
Person1   40     180
Person2   52     189
Person3   47     158
Person4   19     197
Person5   17     160
```

We need to specify the columns first, and then the rows.

Slicing rules for lists still apply.

```
>>> PeopleDF[['HEIGHT','AGE']]['Person1':'Person4']
         HEIGHT  AGE
Person1     194   36
Person2     158   74
Person3     193   23
Person4     155   56
```

Accessing and manipulating data

- Manipulating values

Original DataFrame:

```
         AGE  HEIGHT
Person1   40     180
Person2   52     189
Person3   47     158
Person4   19     197
Person5   17     160
```

- `>>> df2 = df.copy()` Is useful if we want to modify entries without affecting the original DataFrame.

- Python arithmetic functions can be applied to Pandas Series, and most NumPy operations are also supported:

```
>>> PeopleDF2 = PeopleDF.copy()
... PeopleDF2['HEIGHT_IN_M']=PeopleDF['HEIGHT']/100
... print(PeopleDF2.head())
         AGE  HEIGHT  HEIGHT_IN_M
Person1   40     180         1.80
Person2   52     189         1.89
Person3   47     158         1.58
Person4   19     197         1.97
Person5   17     160         1.60
```

Accessing and manipulating data

- Manipulating values: Apply

Original DataFrame:

```
        AGE  HEIGHT
Person1  40     180
Person2  52     189
Person3  47     158
Person4  19     197
Person5  17     160
```

- Apply function allow for more complex data manipulation with series:

```
>>> PeopleDF2['ADULT'] = PeopleDF2['AGE'].apply(lambda val: True if val>18 else False)
... print(PeopleDF2.head())
         AGE  HEIGHT  HEIGHT_IN_M  ADULT
Person1   40     180         1.80   True
Person2   52     189         1.89   True
Person3   47     158         1.58   True
Person4   19     197         1.97   True
Person5   17     160         1.60  False
```

- We can alsouse apply with custom functions:

```
>>> def BirthYear(x):
...     return 2021-x
... PeopleDF2['BIRTHYEAR'] = PeopleDF2['AGE'].apply(BirthYear)
... print(PeopleDF2.head())
         AGE  HEIGHT  HEIGHT_IN_M  ADULT  BIRTHYEAR
Person1   40     180         1.80   True       1981
Person2   52     189         1.89   True       1969
Person3   47     158         1.58   True       1974
Person4   19     197         1.97   True       2002
Person5   17     160         1.60  False       2004
```

Accessing and manipulating data

- Manipulating values: Apply

Original DataFrame:

	AGE	HEIGHT
Person1	40	180
Person2	52	189
Person3	47	158
Person4	19	197
Person5	17	160

- Apply function can also be applied to a DataFrame, and we can specify if we want to use columns or rows:

```
>>> def countmissing(x):
...     return sum(x.isnull())
... #Applying per column:
... print(PeopleDF2.apply(countmissing, axis=0))
... #Applying per row:
... print(PeopleDF2.apply(countmissing, axis=1).head())
AGE            0
HEIGHT         0
HEIGHT_IN_M    0
ADULT          0
BIRTHYEAR      0
dtype: int64
Person1    0
Person2    0
Person3    0
Person4    0
Person5    0
dtype: int64
```

Accessing and manipulating data

- Manipulating values: Map

Original DataFrame:

```
         AGE  HEIGHT
Person1   40     180
Person2   52     189
Person3   47     158
Person4   19     197
Person5   17     160
```

- The `>>> pd.Series.map()` function can be used for substituting each value in a Series with another value, that may be derived from a function, a dictionary or another Series.

```
>>> PeopleDF2['ADULT'] = PeopleDF2['ADULT'].map({True: 'Yes', False: 'No'})
... print(PeopleDF2.head())
         AGE  HEIGHT  HEIGHT_IN_M ADULT  BIRTHYEAR
Person1   40     180         1.80   Yes       1981
Person2   52     189         1.89   Yes       1969
Person3   47     158         1.58   Yes       1974
Person4   19     197         1.97   Yes       2002
Person5   17     160         1.60    No       2004
```

Accessing and manipulating data

- Manipulating values: Filtering data

  - In the following example values are filtered according to boolean expressions:

```
>>> FilteredDF = PeopleDF2.loc[(PeopleDF2['ADULT']=='No') & (PeopleDF2['HEIGHT']<180)]
... print(FilteredDF)
          AGE  HEIGHT  HEIGHT_IN_M ADULT  BIRTHYEAR
Person5    17     160         1.60    No       2004
Person6    17     161         1.61    No       2004
Person16   13     171         1.71    No       2008
Person17   15     156         1.56    No       2006
Person18   18     160         1.60    No       2003
```

```
... print(FilteredDF.index)
Index(['Person5', 'Person6', 'Person16', 'Person17', 'Person18'], dtype='object')
```

```
>>> FilteredDF =FilteredDF.reset_index()
>>> FilteredDF.drop(columns=['index'])
   AGE  HEIGHT  HEIGHT_IN_M ADULT  BIRTHYEAR
0   17     160         1.60    No       2004
1   17     161         1.61    No       2004
2   13     171         1.71    No       2008
3   15     156         1.56    No       2006
4   18     160         1.60    No       2003
```

Useful Pandas methods

- `>>> pd.DataFrame.describe()` returns descriptive statistics of the data in a Pandas DataFrame or Series.

```
>>> PeopleDF2.describe()
             AGE      HEIGHT  HEIGHT_IN_M    BIRTHYEAR
count  20.000000   20.000000    20.000000    20.000000
mean   41.350000  171.300000     1.713000  1979.650000
std    22.269344   16.939832     0.169398    22.269344
min    13.000000  150.000000     1.500000  1944.000000
25%    17.750000  158.000000     1.580000  1959.500000
50%    38.000000  161.500000     1.615000  1983.000000
75%    61.500000  189.000000     1.890000  2003.250000
max    77.000000  198.000000     1.980000  2008.000000
```

- `>>> astype()` is used to cast a Python object to a particular data type

- `>>> to_datetime()` converts a Python object to datetime format. It can take an integer, floating point number, list, Pandas Series, or Pandas DataFrame as argument.

Useful Pandas methods

- `>>> value_counts()` returns a Pandas Series containing the counts of unique values.

- `>>> drop_duplicates()` returns a Pandas DataFrame with duplicate rows removed. Even among duplicates, there is an option to keep the first occurrence (record) of the duplicate or the last.

- `>>> sort_values()` sorts a Series or DataFrame by values in ascending or descending order. By specifying the inplace attribute as True, you can make a change directly in the original DataFrame.

- `>>> WeatherDF['TEMP'].fillna(24, inplace=True)` helps to replace all NaN values in a DataFrame or Series by imputing these missing values with appropriate values.

Aggregation functions

Pandas has a number of aggregating functions that reduce the dimension of the grouped object. We can then use groupby() to split the DataFrame into groups. It is similar to a SQL database.

- mean(): Compute mean of groups
- sum(): Compute sum of group values
- size(): Compute group sizes
- count(): Compute count of group
- std(): Standard deviation of groups
- var(): Compute variance of groups
- sem(): Standard error of the mean of groups
- describe(): Generates descriptive statistics
- first(): Compute first of group values
- last(): Compute last of group values
- nth() : Take nth value, or a subset if n is a list
- min(): Compute min of group values
- max(): Compute max of group values

```
>>> PeopleDF2.groupby('BIRTHYEAR').mean()
            AGE  HEIGHT  HEIGHT_IN_M
BIRTHYEAR
1944       77.0   193.0        1.930
1951       70.0   151.0        1.510
1952       69.0   161.0        1.610
1954       67.0   158.0        1.580
1955       66.0   189.0        1.890
1961       60.0   162.0        1.620
1962       59.0   155.0        1.550
1969       52.0   189.0        1.890
1974       47.0   158.0        1.580
1981       40.0   180.0        1.800
1985       36.0   150.0        1.500
1986       35.0   193.0        1.930
1987       34.0   198.0        1.980
2002       19.0   197.0        1.970
2003       18.0   160.0        1.600
2004       17.0   160.5        1.605
2005       16.0   184.0        1.840
2006       15.0   156.0        1.560
2008       13.0   171.0        1.710
```

Combining data: merge, join, concat

- **concat():** used for combining DataFrames across rows or columns.

```
>>> Result = pd.concat([df1,df4])
```

df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df4

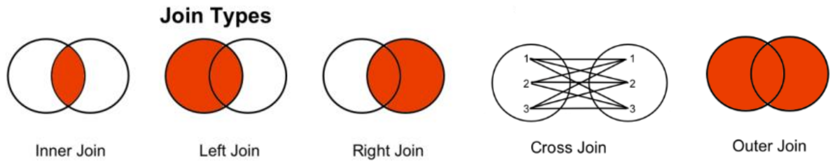
	B	D	F
2	B2	D2	F2
3	B3	D3	F3
6	B6	D6	F6
7	B7	D7	F7

Result

	A	B	C	D	B	D	F
0	A0	B0	C0	D0	NaN	NaN	NaN
1	A1	B1	C1	D1	NaN	NaN	NaN
2	A2	B2	C2	D2	B2	D2	F2
3	A3	B3	C3	D3	B3	D3	F3
6	NaN	NaN	NaN	NaN	B6	D6	F6
7	NaN	NaN	NaN	NaN	B7	D7	F7

Combining data: merge, join, concat

- **merge():** for combining data on common columns or indices.
    - When using merge, we provide a left and a right DataFrame.
    - Additional arguments define how they are merged:
        - How: *{'left', 'right', 'outer', 'inner', 'cross'}*
        - On: Column or index level names to join on.
        - …



Combining data: merge, join, concat

- **.join():** for combining data on a key column or an index
  - While merge() is a module function, .join() is an object function.
  - It uses merge under the hood, but the only required parameter is the other DataFrame we want to join.
  - As in the previous case, additional parameters define how they are joined:
    - other: The other DataFrame to be joined.
    - on: They column or index that will be taken as common. Default is None
    - how: This has the same options as how from merge(). The difference is that it is index-based unless you also specify columns with on.…

- This topic includes many cases and it is better understood with some hands-on code:
  https://pandas.pydata.org/pandas-docs/stable/user_guide/merging.html

Conclusions

- **Pandas** is a great tool to handle diverse types data and relational **databases**.
- **Series** and **DataFrames** are the basic objects. They are flexible data structures that can storage different kind of data.
- **Pandas** support most of **NumPy** functionalities that can be applied to Series.
- **Pandas** can be integrated with machine learning libraries like **sklearn**.