Digital Technology

Pandas

POLITECNICO
MILANO 1863

Carlo Bono

carlo.bono@polimi.it

Pandas

Pandas is the de facto standard package for data manipulation and exploratory data analysis

Plenty of functions for:

- data loading and writing
- data manipulation
- data aggregation
- data analysis
- data cleaning

Pandas

Pandas is integrated with popular Python data science packages like:

NumPy for numerical computing

Matplotlib, Seaborn, Plotly for data visualization

scikit-learn for machine learning

Pandas

Pandas is a data manipulation package in Python for tabular data.

Tabular data: rows and columns!
They are called DataFrames in pandas

Intuitively, you can think of a DataFrame as a database table, an excel, spreadsheet

Under the hood, a DataFrame is a dictionary of Series… A special Pandas object that is basically a typed list (shares some similarities with a Python list, but it's a different data type)

Pandas

Typical operations:

- Open datasets from many sources: databases, APIs, spreadsheets, CSV files…
- Clean datasets, e.g. fill missing values, remove “bad” records
- Tidy up data, reshape data format
- Group data and aggregate it, extracting summary statistics
- Visualize data
- Analyze time series and text data

Firing up pandas

Install it (not needed on Colab)

```
pip install pandas
!pip install pandas # in a Jupyter notebook
```

Import it

```
import pandas as pd
```

Initialize a DataFrame

```
df = pd.DataFrame(
        {"Name": ["Alice", "Bob", "Charlie", "Tom"],
        "Age": [25, 30, 35, 27] })
```

Read data

Open data contained in a CSV file:

```
df = pd.read_csv("myfile.csv")
```

Comma-separated values are just files in which fields are separated by… Well, could be commas, could be some other field.

Moreover, they usually need to “quote” text values (e.g. with double quotes for enclosing it) since they could contain commas – or the separator

Read data

Open data contained in a TSV file:

```
df = pd.read_csv("myfile.tsv", sep="\t")
```

Or a somewhat separated file

```
df = pd.read_csv("myfile.csv", sep="\s")
```

Read data

Open data contained in an excel file:

```
pd.read_excel("myfile.xlsx")
```

Open data contained in an excel file, specific worksheet by index:

```
pd.read_excel("myfile.xlsx", sheet_name=0)
```

By name:

```
pd.read_excel("myfile.xlsx", sheet_name="MySheet")
```

Read data

Open data contained in a JSON file:

```
df = pd.read_json("myfile.json")
```

Yes, JSON files do *not* have a tabular format… But they can be flattened, or processed

Write data

Writing is as simple as:

```
df.to_csv("out.csv", index=False)
```

The ‘index’ parameter decides if we want to output the index as a column, or not

```
df.to_csv("out.txt", header=df.columns, index=None, sep=" ")
```

Inspect data

After data is loaded, you can have a peek:

```
df.head()
```

or

```
df.tail()
```

Or print summary statistics (count, mean, standard deviation, range, and quartiles) of numeric columns

```
df.describe()
```

Inspect data

Get information about the data types and memory size:

```
df.info(show_counts=True, memory_usage=True, verbose=True)
```

Get the shape of data (rows, columns), without parentheses! It is an attribute, not a function… Pandas is a trickster

```
df.shape
```

Inspect data

Check whether each element in a DataFrame is missing / empty

```
df.isnull()
```

Get the number of empty fields, by column

```
df2.isnull().sum()
```

Select data

You can get a column as a pandas Series square brackets [‘xxx’] with a column name in it

```
df["colname1"]
df['colname1']
df.colname1
```

For more than one column, use a list:

```
df[["col1", "col2"]]
```

Select data

You can get a row in pandas using the index value or the index location.

For the row with index 1:

```
df.loc[1]
```

For the row at location 1:

```
df.iloc[1]
```

Select data

You can also select multiple rows using slices, in the same way:

```
df.loc[0:100]
df.iloc[0:100]
```

Or select by lists:

```
df.loc[ [ 0, 10 ] ]
```

Select data

pandas lets you filter data by condition. A condition returns a series with Boolean values for the condition. You can then use this series for subsetting data:

```
my_boolean_filter = df.field1 == 1
df[ my_boolean_filter ]
```

or in a more compact way:

```
df[ df.field1 == 1  ]
```

Or even combining conditions and column selection:

```
df.loc[ df["col1"] > 100, ["col1", "col2", "col3"] ]
```

Clean data

You can drop every column with null values like this:

```
df = df.dropna(axis=1)
```

Or inplace (without reassigning the dataframe):

```
df.dropna(inplace=True, axis=1)
```

Or remove records with one or more missing values:

```
df.dropna(inplace=True, how="all")
```

Clean data

You can replace “missing” values selectively:

```
meanval = df["col1"].mean()

df = df["col1"].fillna(meanval)
```

Or on all the data frame:

```
df = df.fillna(0.)
```

Clean data

You can remove duplicated entries:

```
df.drop_duplicates()
```

You can also specify on which columns the duplicated values should be evaluated, and if to keep the first or last values (depends on the ordering, but you can always change the ordering…)

Clean data

Renaming columns:

```
df.rename(columns = {"MyOldName": "MyNewName"})
```

Or directly assign column names:

```
df.columns = ["A", "B"]
```

Drop columns:

```
df.drop(["A", "B"]
)
my_df = my_df.drop(['Money'], axis=1)
```

Analyzing data

Calculating values:

```
df["ratio"] = df["A"] / df["B"]
```

Counting occurrences:

```
df.mycolumn.value_counts()
```

Analyzing data

Simple aggregations:

```
df.mean()
df.median()
df.mode()
```

Counting occurrences:

```
df.mycolumn.value_counts()
```

Analyzing data

Aggregations can also be computed per group:

```
df.groupby("A").mean()
```

And with multiple groups:

```
df.groupby(["A", "B"]).mean()
```

Analyzing data

Pandas also supports many data visualizations…

See Colab in the course materials