

# DevOps Crash Course



by Oleksii Yakivchik

**softserve**

# Python package management

- Package manager – pip
- Install package – pip install
- Remove package – pip uninstall
- List packages – pip list
- Upgrade pip – pip install --upgrade pip

# Python virtual environment

Python applications will often use packages and modules that don't come as part of the standard library. Applications will sometimes need a specific version of a library, because the application may require that a particular bug has been fixed or the application may be written using an obsolete version of the library's interface.

# Python virtual environment

The solution for this problem is to create a virtual environment, a self-contained directory tree that contains a Python installation for a particular version of Python, plus a number of additional packages.

Different applications can then use different virtual environments. To resolve the earlier example of conflicting requirements, application A can have its own virtual environment with version 1.0 installed while application B has another virtual environment with version 2.0. If application B requires a library be upgraded to version 3.0, this will not affect application A's environment.

# Python virtual environment

The module used to create and manage virtual environments is called venv. venv will usually install the most recent version of Python that you have available. If you have multiple versions of Python on your system, you can select a specific Python version by running python3 or whichever version you want. The next command will create new virtual env “tutorial-env”

```
python3 -m venv tutorial-env
```

# Python virtual environment

To start work with the new virtual environment, use the next command:

```
source tutorial-env/bin/activate
```

Now you can start installing packages for your project.

For portability purposes, you should have the requirements.txt file with all your dependencies. To create this file with packages from your virtual environment, you can use the next command:

```
pip freeze > requirements.txt
```

# Pandas

Pandas is an open-source, BSD-licensed Python library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language. Python with Pandas is used in a wide range of fields including academic and commercial domains including finance, economics, Statistics, analytics, etc.

# Pandas features

- A fast and efficient DataFrame object for data manipulation with integrated indexing;
- Tools for reading and writing data between in-memory data structures and different formats: CSV and text files, Microsoft Excel, SQL databases, and the fast HDF5 format;
- Intelligent data alignment and integrated handling of missing data: gain automatic label-based alignment in computations and easily manipulate messy data into an orderly form;
- Flexible reshaping and pivoting of data sets;
- Intelligent label-based slicing, fancy indexing, and subsetting of large data sets;

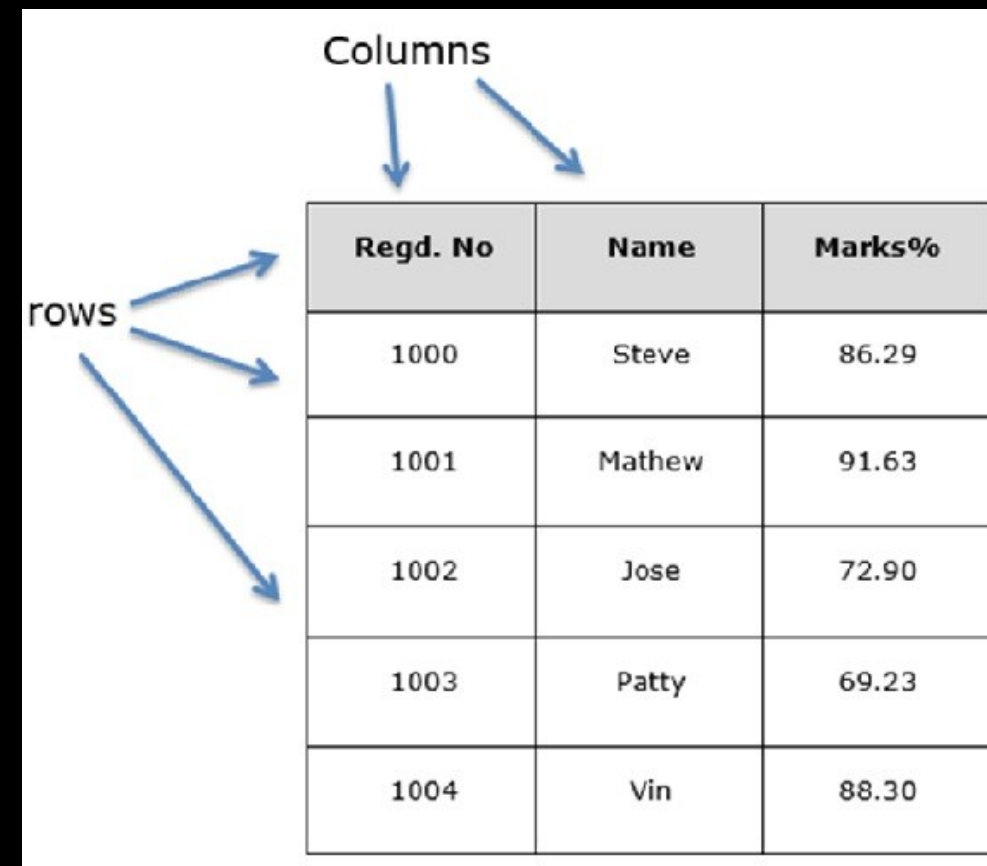


# Pandas features

- Columns can be inserted and deleted from data structures for size mutability;
- Aggregating or transforming data with a powerful group by engine allowing split-apply-combine operations on data sets;
- High performance merging and joining of data sets;
- Hierarchical axis indexing provides an intuitive way of working with high-dimensional data in a lower-dimensional data structure;
- Time series-functionality: date range generation and frequency conversion, moving window statistics, date shifting and lagging. Even create domain-specific time offsets and join time series without losing data.

# Pandas DataFrame

A Data frame is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns.



The diagram illustrates a Pandas DataFrame as a table with three columns and five rows. The columns are labeled 'Regd. No', 'Name', and 'Marks%'. The rows contain data for five individuals: Steve, Mathew, Jose, Patty, and Vin. Blue arrows point from the labels 'Columns' and 'rows' to their respective parts of the table.

Columns		
Regd. No	Name	Marks%
1000	Steve	86.29
1001	Mathew	91.63
1002	Jose	72.90
1003	Patty	69.23
1004	Vin	88.30

rows

# Features of DataFrame

- Potentially columns are of different types
- Size – Mutable
- Labeled axes (rows and columns)
- Can Perform Arithmetic operations on rows and columns

# Create DataFrame

A pandas DataFrame can be created using various inputs like:

- Lists
- dict
- Series
- Numpy ndarrays
- Another DataFrame

# Logparser

Let's write our *logparser* script.

Usage of our script:

```
python3 logparser.py <path-to-logfile>
```

We should install pandas for our script:

```
pip install pandas
```

# Logparser

Our script should take the path to the log file as a command-line argument:

```
import sys
```

```
file = open(sys.argv[1], 'r')
```

# Logparser

Now we should parse our logs using regular expression line by line and store parsed logs in list:

```
logs_entries = []
```

```
for entry in file:
```

```
    line = re.compile(r'^((Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|  
Nov|Dec)\s+\d{1,2}\s+\d{2}:\d{2}:\d{2})\s+(\S+)\s+(sshd)\S+:\s+(.*?(\  
d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}).*)$').search(entry)
```

```
    if line:
```

```
        ...
```

# Logparser

```
...
datetime_str = line.group(1) + " " +
str(datetime.datetime.now().year)
    datetime_obj = datetime.datetime.strptime(datetime_str,
'%b %d %H:%M:%S %Y')
    logs_entries.append({"hostname": line.group(3),
"ip_address": line.group(6), "date_time": datetime_obj,
"message": line.group(5)})
```



# Logparser

Now we should close the file and write collected results in Excel file:

```
file.close()  
df = pd.DataFrame(logs_entries)  
df.to_excel("output/access_logs.xlsx")
```

# Logparser

If you want to format your document through deleting duplicating rows in some columns, you can do it this way:

```
df["hostname"].mask(df["hostname"].duplicated(),  
inplace=True)  
df["ip_address"].mask(df["ip_address"].duplicated(),  
inplace=True)
```