

6 – Data Wrangling libraries in Python – Pandas

Nicolas Pfeuffer

Table of Contents

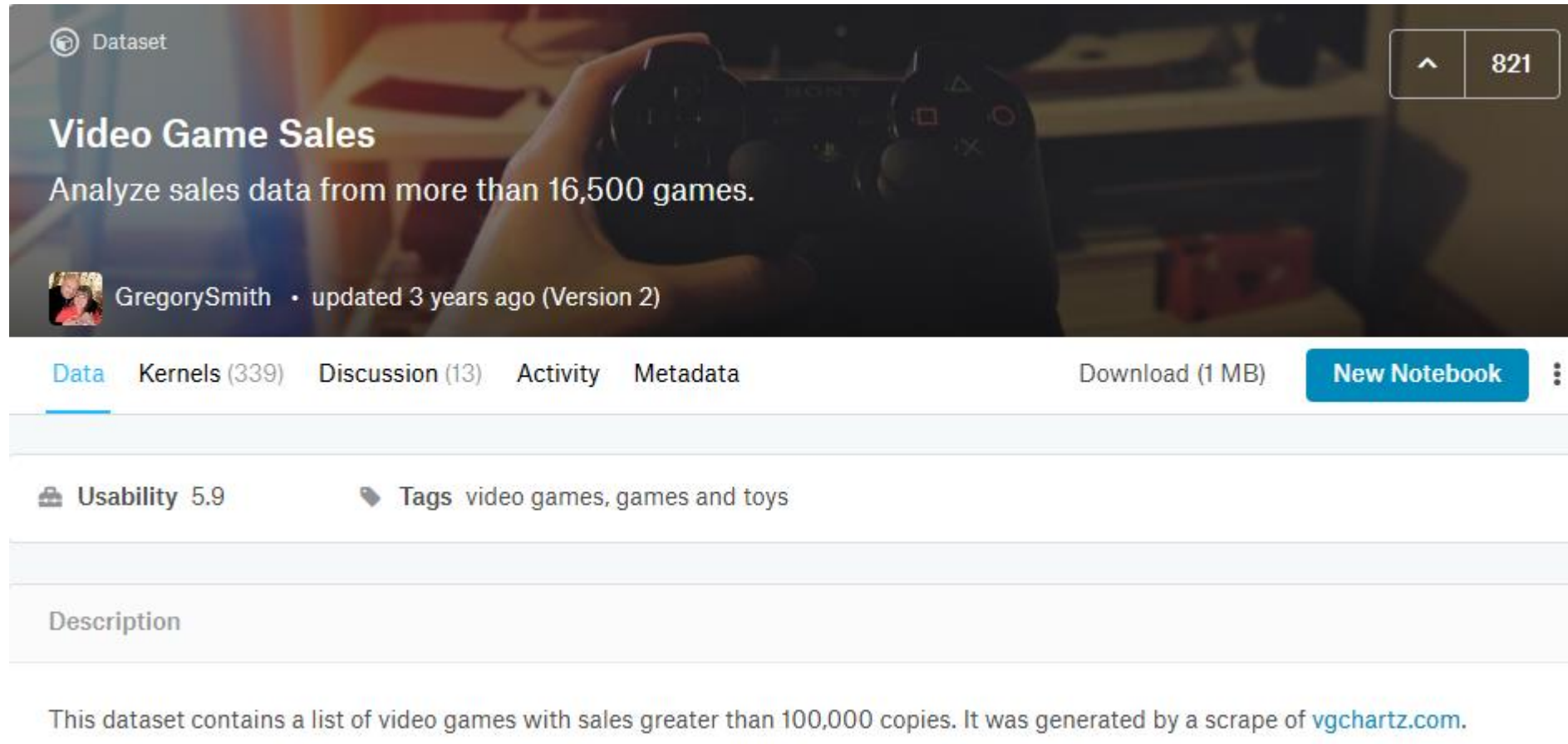
- 1. Using whole Datasets for Data Analysis**
- 2. Intro to pandas - pd**
- 3. Descriptive Statistics with pandas**

Using large Datasets for Data Analysis

- Until now, you have learned how to:
 - Assign single variables to certain datatypes (int, float, string)
 - Structure data through lists, sets, dictionaries
 - form control structures and loops
 - Construct functions
 - Apply the python standard libraries, especially importing them and reading in files
 - Got to know NumPy, an important numerical library for data analysis
- It is time that we take the next step: reading in large datasets
- For this cause, we will utilize a dataset on video game sales from kaggle.com

Using large Datasets for Data Analysis

- For this cause, we will utilize a dataset on video game sales from kaggle.com



The screenshot shows the Kaggle dataset page for 'Video Game Sales'. At the top, it says 'Dataset' with a circular icon. The title 'Video Game Sales' is prominently displayed, followed by the subtitle 'Analyze sales data from more than 16,500 games.' Below this, the creator's name 'GregorySmith' and the update information 'updated 3 years ago (Version 2)' are shown. A navigation bar includes links for 'Data', 'Kernels (339)', 'Discussion (13)', 'Activity', and 'Metadata', along with a 'Download (1 MB)' button and a 'New Notebook' button. The 'Usability' score is 5.9, and the tags are 'video games, games and toys'. The 'Description' section states: 'This dataset contains a list of video games with sales greater than 100,000 copies. It was generated by a scrape of vgchartz.com.'

- Do not worry about the scraping part. For now, the dataset is provided for you in your workspace.

NumPy: Lessons learned

- Numpy provides a plethora of fast, mathematical functions
- These functions are typically performed on ndArrays.
- ndArrays are computationally efficient and also referred to as scalars, vectors or matrices.
- The Pandas package and DataFrame inherit many of the positive features from NumPy!

File reading – the old fashioned way

- First use the os package to list all the files in your directory
- If you can not find it, change the directory via os commands or the directory browser on the top right.
- You should find a file called “vgsales.csv”
- Try to read this file in via the csv.reader and print the first 10 rows:

```
with open("test.csv", "r") as testFile:
    reader = csv.reader(testFile, delimiter=",")
    rows = list(reader)
    for i in range(10): #insert the number of rows to print
        print(rows[i])
```

File reading – the old fashioned way

- What now?
 - For data science workflows, many statistical and mathematical operations need to be performed
 - Simple operations are possible, but the file must stay opened and will be closed after the operations
 - Could store the file in nested lists or dictionaries: complex and computationally inefficient
- It seems that the data structures and operations we learned until now are not sufficient for data science workflows on large datasets
- Pandas package provides a solution

Table of Contents

1. Using whole Datasets in for Data Analysis

2. Intro to pandas - pd

3. Descriptive Statistics with pandas

DataFrames – the Pandas way

- Pandas is a comprehensible and powerful library for data analysis
- Allows us to load files of various file types into a data structure called DataFrame
- DataFrame:
 - persistent data structure
 - represent data in a tabular way
 - consists of rows and columns
 - Columns are called 'Series' and are a subordinate data structure
 - Rows contain the actual Datapoint/ Rows

DataFrames – the Pandas way

- Take a look at a DataFrame

Column/ Series /Attribute/ Feature

↓

Datapoint/ Row →

ID	Name	Salary
0	TestGuy	2000
1	James	2333
2	Alan	3000
3	Ada	3000

DataFrame & Pandas – A first look

- Initialize a pandas dataframe. The general naming convention for dataframes is “df”.

```
df = pd.read_csv('test.csv', sep=',')
```

- First, take a look at the dataframe in the variable explorer.
- Now, let's see how many functions pandas provides. Then, how many functions Pandas provides for the df especially.

```
print(dir(pd))  
print(dir(pd.DataFrame))
```

- Lots of helpful functions!
- The functions we are going to go through will empower you to:
 - Get an overview over the dataset
 - Get descriptive statistics on the dataset
 - Find first indices for correlations in the dataset

DataFrames – the Pandas way

- Print the columns

```
print(df.columns)
```

Column/ Series /Attribute/ Feature

↓

	ID	Name	Salary
Datapoint/ Row →	0	TestGuy	2000
	1	James	2333
	2	Alan	3000
	3	Ada	3000

```
Columns["ID", "Name", "Salary"]
```

DataFrames – the Pandas way

- Print the column length. Similar to python lists!

```
print(len(df.columns))
```

Column/ Series /Attribute/ Feature



Datapoint/ Row



ID	Name	Salary
0	TestGuy	2000
1	James	2333
2	Alan	3000
3	Ada	3000

3

DataFrames – the Pandas way

- The shape may provide us with more information.

```
df_shape = df.shape
print(df_shape)
```

Column/ Series /Attribute/ Feature



Datapoint/ Row



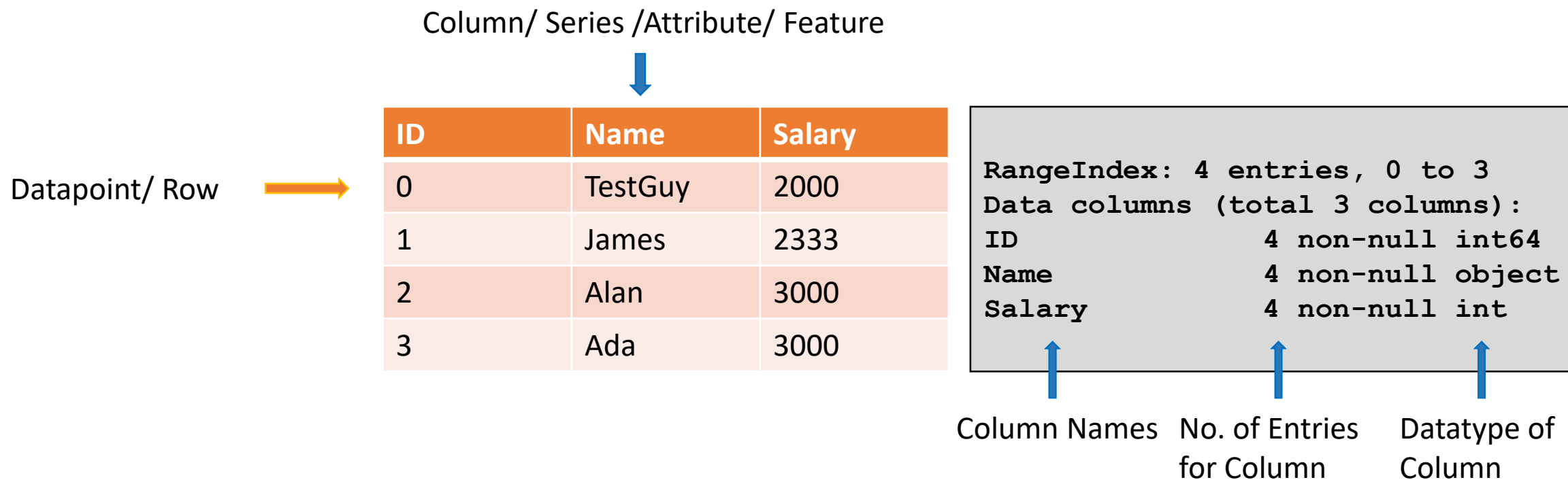
ID	Name	Salary
0	TestGuy	2000
1	James	2333
2	Alan	3000
3	Ada	3000

(4, 3)

DataFrames – the Pandas way

- Let's get much more information! We want to see the datatypes, etc...

```
df_info = df.info()
print(df_info)
```



DataFrame Datatypes

- In your very first python session, you learned about primitive datatypes, such as int, string, float.
- Complex datatypes are almost always represented as objects. Let's print the column 'Name' with the object datatype, to see if it is really complex and why it is labeled as object.
- Attention! There are two obvious ways to do this.

```
print(df.Name)
print(df['Name'])
```

- As we can see, the object datatype actually contains strings.
- Pandas initially tries to assign datatypes to the various columns.
- Sometimes it does not get the correct type. See the Documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/basics.html#dtypes
- Citation: *Pandas uses the object dtype or StringDtype for storing strings.*

DataFrame Datatypes

- Problem solved. But what about Year? Should it not be int? Print the column.

```
print(df.Year)
print(df['Year'])
```

- It should be int, yet it has been assigned float values.
- Again, we find our answer in the pandas documentation, see: http://pandas-docs.github.io/pandas-docs-travis/user_guide/missing_data.html#integer-dtypes-and-missing-data
- Citation: *Because NaN (missing value) is a float, a column of integers with even one missing values is cast to floating-point dtype (see Support for integer NA for more).*

Missing values alter the datatype to float64!

- Missing values are reflected in the entries and datatypes. Let's see:

```
df_info = df.info()
print(df_info)
```

Column/ Series /Attribute/ Feature

Datapoint/ Row

ID	Name	Salary
0	TestGuy	2000
1	James	2333
2	Alan	
3	Ada	3000

RangeIndex: 4 entries, 0 to 3

Data columns (total 3 columns):

ID 4 non-null int64

Name 4 non-null object

Salary 3 non-null float64

Column Names No. of Entries for Column Datatype of Column

Missing values nach indizierung und slice schieben

DataFrame – Handling missing values

- This explains the issue. Let's check for missing values if it is true for the dataset.
- Try to find the function, which shows all null values. Then apply the `.sum()` function on top of it, to show the sum of null values per column.

```
df_null_values = df.isnull().sum()  
print(df_null_values)
```

- Indeed! We have 271 missing values in the Year column, as well as some in the publisher column.
- We could now decide to drop all the rows, where information is missing, try to fill it with sample data (not very useful in our case) or drop the column (also not useful).
- Since we have a lot of data, we will drop the rows where year and publisher data is missing. Use **`pd.DataFrame.dropna`** on our **`dataframe`** to drop the rows. Then print the info on our dataset again.

```
df = df.dropna(axis=0)  
df.info()
```

DataFrame – Handling missing values

- The 306 records have been dropped, such that we now only have 16291 records.
- It appears that Year is still a float. Let's convert it to int like this: **`df['columnName'].astype('int64')`**
- See: https://pandas.pydata.org/pandas-docs/stable/getting_started/basics.html#astype
- Then print info again.

```
df['Year'] = df['Year'].astype('int64')  
df.info()
```

- Great job! You have successfully cleaned your data from rows with missing values and converted a column to its correct datatype.
- Let's start with some manipulation and analysis.

DataFrame – Manipulation & Data Analysis

- Let's see what the top rows and the bottom rows look like.

```
df.head()  
df.tail()
```

- Get access to the 55 record. There are two options.

```
df.loc[54]  
df.iloc[54]
```

- While loc searches the index by named labels (such as strings, but also int), iloc searches for row number.
- The differentiation of these two will become much clearer when performing slicing operations.

DataFrame – loc & iloc

- Remember the slicing operations for strings and lists.
- In the pandas case, the first part of the slice before the comma refers to the records (x-axis), the second to the columns (y-axis): **df.iloc[:x,:y]**
- Let's perform a slice on the rows until index 3, and columns until index 3.

```
df.iloc[:3, :3]
```

- This gives us a 3x3 snapshot of the dataframe, starting at the column 0 (index) and stopping at the column 4 (with index 3). Does it work the same way with loc?

```
df.loc[:3, :3]
```

- Nope. we learned that loc works with 'named' labels. Since the rows do have numeric labels (column 0 contains only numeric values) and include the number 3, the first part should be fine. Let's change the second part to the column label.

```
df.loc[:3, : 'Platform']
```

DataFrame – loc & iloc

- Another example: find the last element 16597 with loc and iloc

```
df.loc[16597]  
df.iloc[16597]
```

- Different story: know that iloc works with indices. We do have a first column that is named index, yet is not consistent with our actual dataset index. This is sometimes native to the dataset, sometimes caused by record drops.
- 16597 would have been the last element. Let's try to get it another way. Remember how it works for lists!

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

- Very good. Now, let's clean up the index with this function.

```
df = df.reset_index(drop=True)
```


Data Analysis in Pandas

- Let's do some advanced stuff now.
- Retain the a subset of the DataFrame containing the records 10-20 (including 10 and 20) and columns Platform - NA_Sales.
- do it with `iloc`.

```
df.iloc[10:21,2:7]
```

- Do it with `loc`.

```
df.loc[10:21, 'Platform': 'NA_Sales']
```

- write the contents of the last operation to a dict.

```
df_slice = df.loc[10:21, 'Platform': 'NA_Sales']  
Dslice = df_slice.to_dict()
```

- Hmm... this does not look quite right. We want the rows to be the keys, not the columns. Let's see what a **df.transpose** can do. Assign the transposed `df_slice` to the appropriate variable. Then write it to the dict.

```
df_slice_transposed = df_slice.transpose()  
Dslice_transposed = df_slice_transposed.to_dict()
```

Data Analysis in Pandas

- Perfect, this is what we wanted. What actually just happened?
- By transposing the dataframe, we changed the columns to rows and the rows to columns. This can sometimes be very helpful in data analysis, for example if you need to convert the data structure or the size of the data structure.
- Especially in Machine Learning and Deep Learning with Neural Nets, being able to transpose data structures is an invaluable feature.

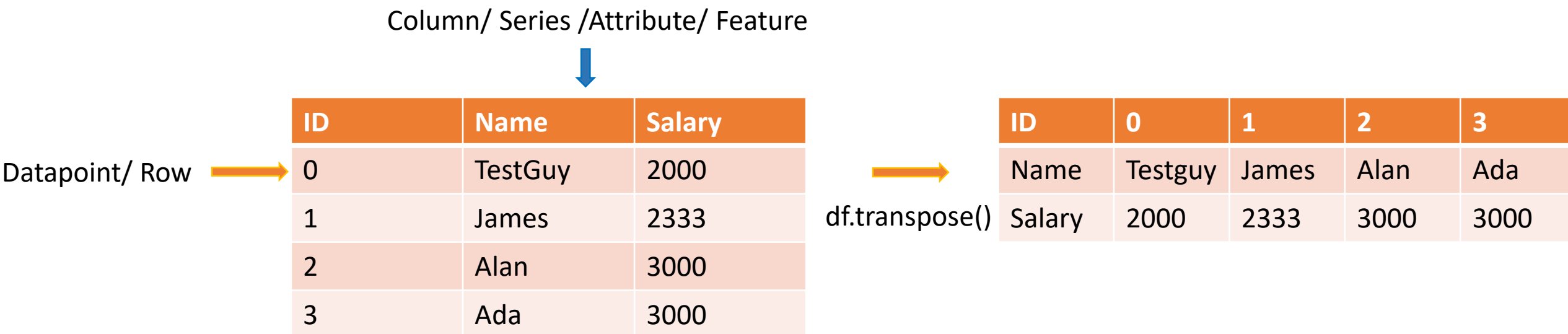


Table of Contents

- 1. Using whole Datasets in for Data Analysis**
- 2. Intro to pandas - pd**
- 3. Descriptive Statistics with pandas**

Descriptive Stats – From np to pd

- Let's take a look at some functions for descriptive stats that numpy provides.

```
a_matmul_arraya.mean()  
a_matmul_arraya.max()  
a_matmul_arraya.min()  
np.quantile(a_matmul_arraya,0.75)  
np.quantile(a_matmul_arraya,0.50)  
np.quantile(a_matmul_arraya,0.25)  
np.std(a_matmul_arraya)
```

- Try to find these yourself in pandas! Apply them in the given order on the pandas df.

Descriptive Stats – Pandas

- Let's take a look at some functions for descriptive stats that numpy provides.

```
df_mean = df.mean()
df_max = df.max()
df_min = df.min()
df_q3 = df.quantile(q=0.75)
df_qmed = df.quantile(q=0.50)
df_q1 = df.quantile(q=0.25)
df_std = df.std()
df_count = df.count()
```

- Try to find these yourself in pandas! Apply them in the given order on the pandas df.

Descriptive Stats – Aggregate Functions

- There are also important aggregate functions that deliver the most interesting desc. stats at once.

```
df_description = df.describe()
```

- To find first hints on how features (another name for our attributes or columns that is used in data analysis) correlate, use the correlation function.

```
df_corr = df.corr()
```

Pandas and Numpy: Lessons learned

- Pandas is probably the most important library for data analysis
- Numpy on the other hand is the most invaluable library for calculus, statistics and math in general
- Both are very important for python data scientists!
- DataFrames are flexible and useful datastructures for working on small and large datasets

References

- <https://pandas.pydata.org/pandas-docs/stable/>
- <https://docs.scipy.org/doc/>
- <https://www.kaggle.com/gregorut/videogamesales/downloads/video-gamesales.zip/2l>

Appendix

Table of Contents

1. Additional info on numPy - np

NumPy: The math library

- A main task in data analysis is calculus and statistics.
- Python itself is already quite powerful, since you can turn mathematical algorithms into code.
- More sophisticated mathematical operations require a lot of code and are often highly standardizable → Numerical Python (NumPy) Library
- linear algebra and statistics
- flexible datastructures for mathematical operations: ndarray

NumPy: The ndArray

- Similar to python lists
- Stores scalars of multiple dimensions:
 - Scalar: 2.25
 - Vector:[1,2,3]
 - Matrix: [[1,2,3],
 [3,4,5]]
- When creating a NumPy Array, every element must be of the same type (different to Python list)
- Fast mathematical computations are possible with the array, between arrays and on the elements of the arrays
- Callable functions from the package at your hand!

NumPy: Python List vs. ndarray

- Initialize a python list and a np.array. The contents should be 1-9

```
List = [1,2,3]  
np.array([1,2,3])
```

- Initialize a multidimensional list and multidimensional array with 3 elements.

```
List = [[1],[2],[3]]  
np.array([1,2,3],  
         [1,2,3],  
         [1,2,3])
```

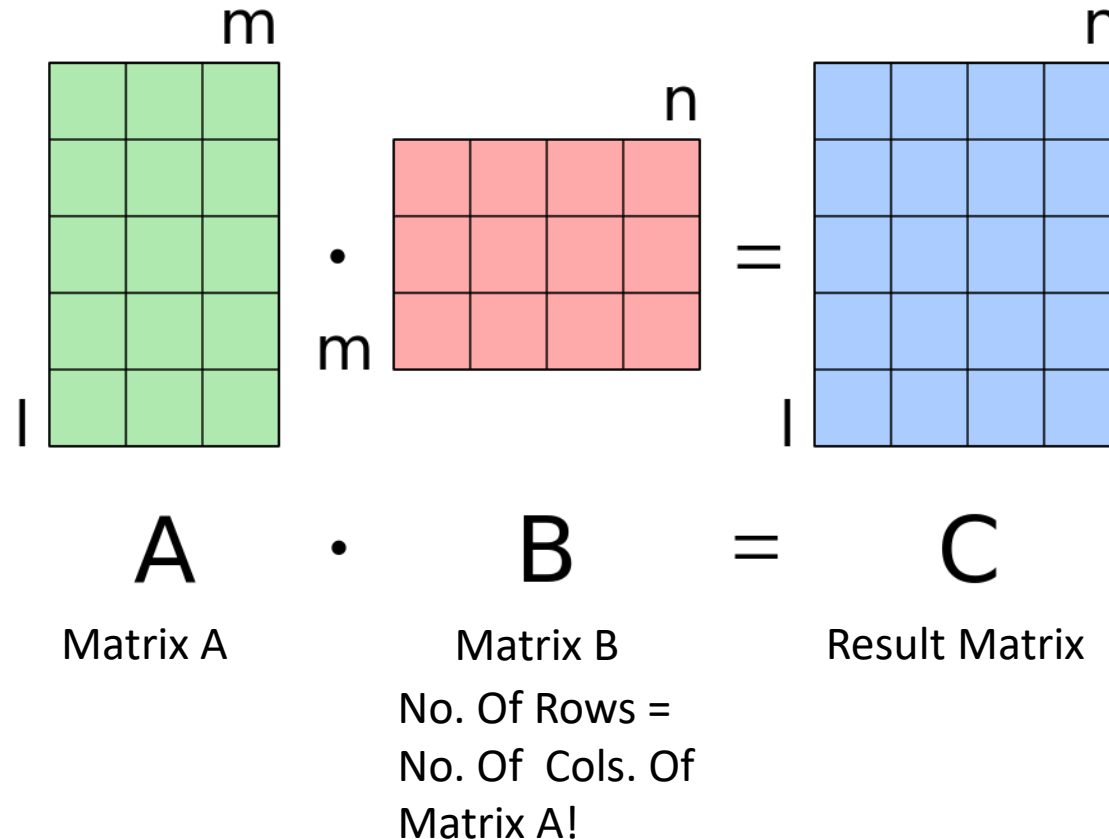
- datatype of numpy arrays is int32, a very basic datatype. This is made so computational efforts are reduced to a minimum.
- elements within lists need to be accessed via nested indexing, elements in arrays are arranged in tables and can be accessed directly. → faster computations.

NumPy: Python List vs. ndarray

- Numpy arrays are also referred to as matrices or vectors, because of their mathematical origin and purpose.
- Hence, mathematical operations can be performed quite easily, fast and on multiple elements.

Matrix multiplications on ndArrays

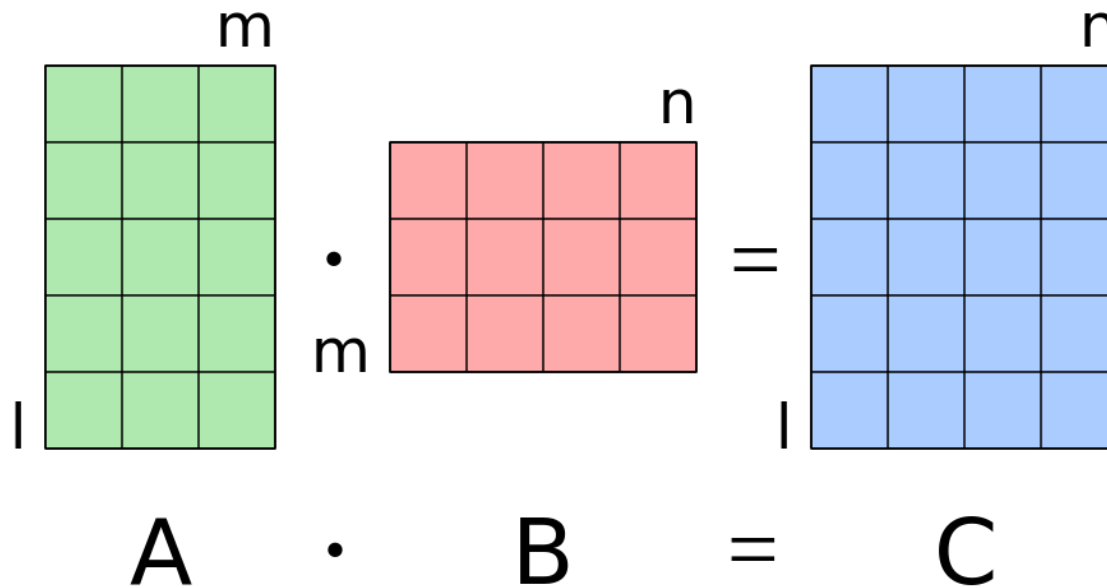
- Can be useful for data analysis.
- Must know for machine learning!
- Think of the typical matrix multiplication.



Matrix multiplications on ndArrays

- Let's perform a matrix multiplication with np.

```
array_C = np.matmul(array_A, array_B)
```



Matrix multiplications on ndArrays

- It appears we have the wrong shape.... but the shape of matrix B is not far from good.

```
ValueError: shapes (3,9) and (3,9) not aligned: 9 (dim 1) != 3 (dim 0)
```

- No. Of Rows of Matrix B = No. Of Cols. Of Matrix A!
- Let's transpose it!

```
array_C = np.matmul(array_A, array_B.transpose())
```

- We can also transpose array_A !

```
array_C2 = np.matmul(array_A.transpose(), array_B)
```

Element-wise operations on ndArrays

- There are also element-wise operations possible.
- Similar to simple python lists, ndarray elements are also accessible by indices.

23	25	48	53
56	21	12	8
87	63	22	99
23	86	34	77
78	21	43	65

C

```
array_c[0,0]  
-> 23
```

Elements in Arrays are indexed by
[row, column]

Element-wise operations on ndArrays

- There are also element-wise operations possible.
- Similar to simple python lists, ndarray elements are also accessible by indices.

23	25	48	53
56	21	12	8
87	63	22	99
23	86	34	77
78	21	43	65

C

```
array_C[2,1]
-> 63
```

Elements in Arrays are indexed by
[row, column]

Element-wise operations on ndArrays

- There are also element-wise operations possible.
- Similar to simple python lists, ndarray elements are also accessible by indices.

23	25	48	53
56	21	12	8
87	63	22	99
23	86	34	77
78	21	43	65

C

```
array_C[3,3]  
-> 77
```

Elements in Arrays are indexed by
[row, column]

Element-wise operations on ndArrays

- We can also not only access, but also modify specific elements.
- Let's multiply it by 3.

23	25	48	53
56	21	12	8
87	63	22	99
23	86	34	77
78	21	43	65

C

```
array_C = array_C[0,0] * 3
```

69	25	48	53
56	21	12	8
87	63	22	99
23	86	34	77
78	21	43	65

C

Element-wise operations on ndArrays

- We can also modify several elements at once very easily.
- Let's multiply all by 3.

23	25	48	53
56	21	12	8
87	63	22	99
23	86	34	77
78	21	43	65

C

```
array_C = array_C * 3
```



69	75	144	159
168	63	36	24
261	189	66	297
69	258	102	231
234	63	129	195

C