Week4: Data Wrangling with Pandas

Chantri Polprasert

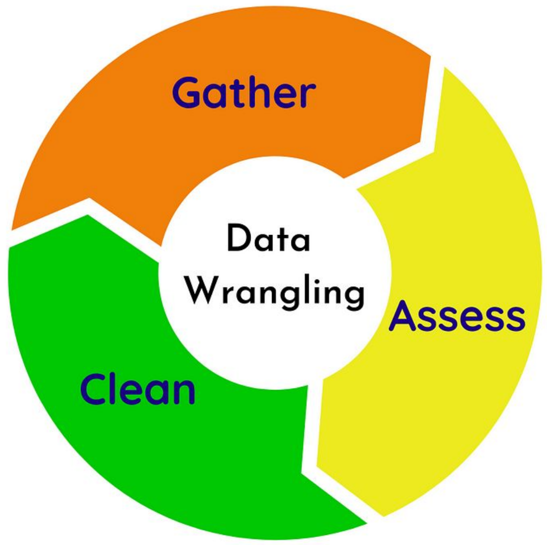
Dept. of ICT,AIT

Agenda

- What is data wrangling?
- Descriptive statistics
- Data cleansing
- Sample data wrangling operations
- Groupby and Concatenation
- String
- Date/time
- Discretization and Binning

Data Wrangling

- Data wrangling, also referred to as **data munging**, is the process of transforming and mapping raw data into a more structured format to make it suitable for analysis.
- The goal is to obtain and assure quality and useful data by **gathering**, **cleaning**, **structuring**, and **assessing** the data



Key Steps in Data Wrangling

1. **Discovering**: Examining the dataset to understand its structure, content, and quality.
2. **Structuring**: Transforming raw data into a suitable format for analysis by categorizing data and standardizing data fields.
3. **Cleaning**: Identifying and correcting errors, inconsistencies, anomalies and inaccuracies within the dataset.
4. **Enriching**: Adding context or new information to the dataset to make it more valuable for analysis.
5. **Validating**: Ensuring the data's accuracy and quality after cleaning, structuring, and enriching.
6. **Publishing**: Preparing the dataset for downstream use and documenting any steps and logic during wrangling.

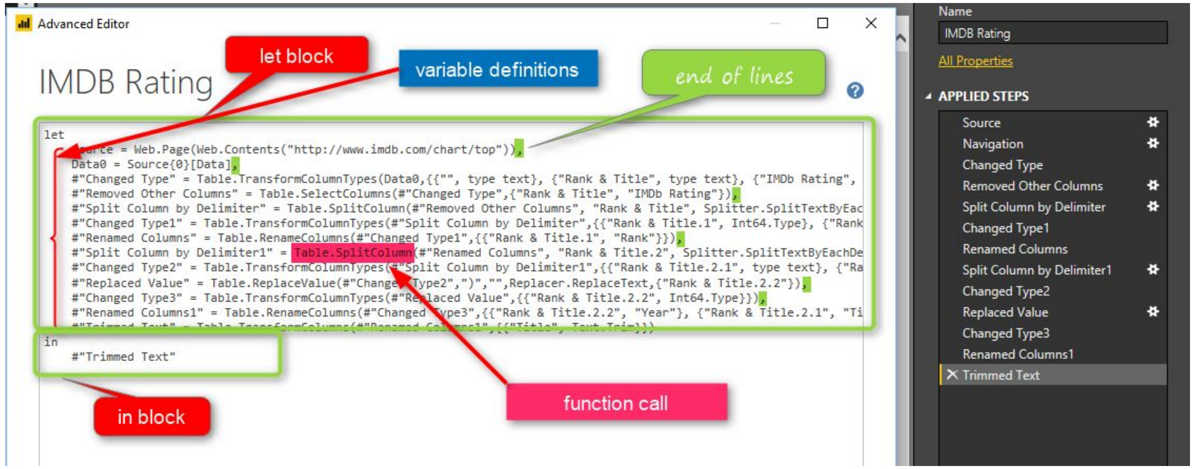
https://www.jaroeducation.com/blog/a-guide-to-the-six-steps-of-data-wrangling/

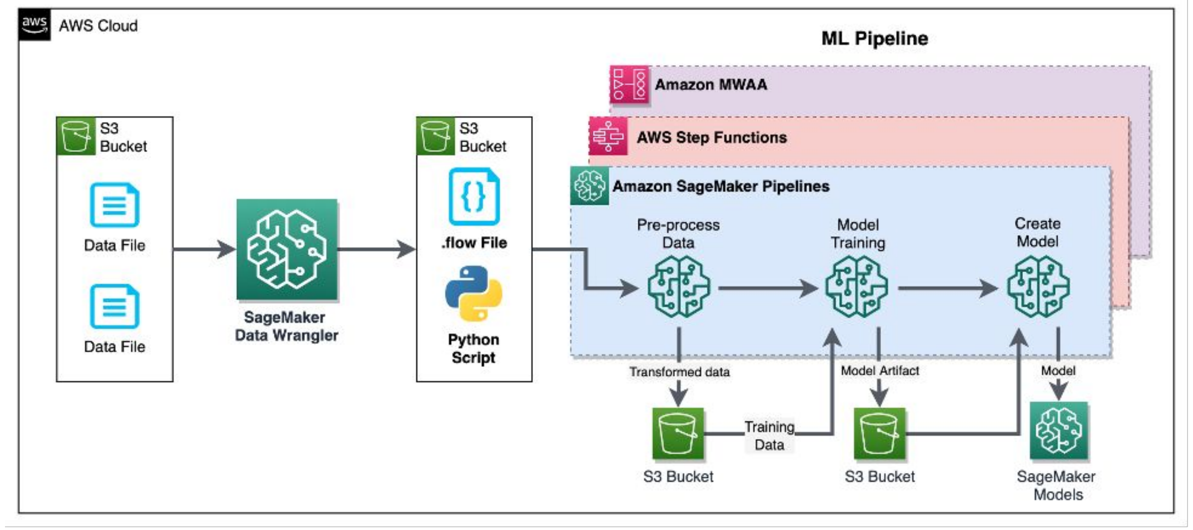
Importance of data wrangling

- **Improves data accuracy and consistency**, maximizing the trustworthiness of insights.
- **Enables easier access and collaboration** by simplifying and organizing data in a consistent manner.
- **Enhances decision-making** by reducing the risk of taking actions based on inaccurate or incomplete information.
- **Allows analysts to analyze more complex data more quickly and achieve more accurate results.**

Examples of Data Wrangling Tools

- Desktop
    - Spreadsheets / Excel Power Query (M formula language)
    - OpenRefine (open source desktop app)
- Cloud based: Amazon SageMaker Data Wrangler (support structured, unstructured, semi-structured data), Google Cloud DataPrep
- Python's Pandas





Pandas

- Pandas is an open-source Python Library providing high-performance data manipulation and analysis tool using its powerful data structures.
- Pandas is one of the most preferred and widely used tools in data munging/wrangling

- Key Features of Pandas
  - Fast and efficient DataFrame object with default and customized indexing.
  - Tools for loading data into in-memory data objects from different file formats.
  - Data alignment and integrated handling of missing data.
  - Reshaping and pivoting of date sets.
  - Label-based slicing, indexing and subsetting of large data sets.
  - Columns from a data structure can be deleted or inserted.
  - Group by data for aggregation and transformations.
  - High performance merging and joining of data.
  - Time Series functionality.

Descriptive Statistics

To explore data

Descriptive Statistics in Pandas

- A large number of methods for computing descriptive statistics on Series and DataFrame
- Most are aggregation (sum(), mean(), quartile(),..) which yield a lower-dimensional result or others (cumsum(), cumprod()) which produce an object of the same size.
- These methods take an axis argument, just like ndarray.{sum, std, …}, but the axis can be specified by name or integer:
  - Series: no axis argument needed
  - DataFrame: “index” (axis=0, default), “columns” (axis=1)
- All such methods have a skipna option signaling whether to exclude missing data (True by default)

A summary table of common functions

Function name	Description
count	Number of non-NA values in the group
sum	Sum of non-NA values
mean	Mean of non-NA values
median	Arithmetic median of non-NA values
std, var	Unbiased (n – 1 denominator) standard deviation and variance
min, max	Minimum and maximum of non-NA values
prod	Product of non-NA values
first, last	First and last non-NA values

Some NumPy methods, like `mean`, `std`, and `sum`, will exclude NAs on Series input by default:

Descriptive Statistics: describe()

- Compute a variety of summary statistics about a Series or the columns of a DataFrame (excluding NAs)

```
print(df)
```

```
    one  two
a  1.40  NaN
b  7.10 -4.5
c   NaN  NaN
d  0.75 -1.3
```

```
df.describe()
```

	one	two
count	3.000000	2.000000
mean	3.083333	-2.900000
std	3.493685	2.262742
min	0.750000	-4.500000
25%	1.075000	-3.700000
50%	1.400000	-2.900000
75%	4.250000	-2.100000
max	7.100000	-1.300000

What about Series with non-numeric value?

Descriptive Statistics: corr()

- ***Syntax:*** *DataFrame.corr(self, method='pearson', min_periods=1)*
- *Ex. df.corr()*
- Computes pairwise Pearson coefficient (ρ) of columns
- Other coefficients available: Kendall, Spearman
  - pearson : standard correlation coefficient
  - kendall : Kendall Tau correlation coefficient
  - spearman : Spearman rank correlation

Covariance

$$\rho_{X,Y} = \frac{\mathrm{cov}(X,Y)}{\sigma_X \sigma_Y}$$

Standard deviation

Descriptive Statistics: corr()

```
DataFrame.corr(method='pearson', min_periods=1, numeric_only=False)
```

Compute pairwise correlation of columns, excluding NA/null values.

method : ***{'pearson', 'kendall', 'spearman'} or callable***

Method of correlation:

- pearson : standard correlation coefficient
- kendall : Kendall Tau correlation coefficient
- spearman : Spearman rank correlation

$$Correlation = \frac{Cov\,(x, y)}{\sigma x * \sigma y}$$

Correlation vs Causation

vs.

- **Correlation** refers to a statistical relationship between two variables, indicating that when one variable changes, the other tends to change as well. (can be positive or negative)
  - **Example**: the relationship between ice cream sales and #of shark attacks.
- **Causation** implies a direct cause-and-effect relationship between two variables. When one variable (the cause) directly affects another variable (the effect), we say that a causal relationship exists.
  - **Example**: smoking and lung cancer.
- Why Correlation Does Not Imply Causation
  - **Third Variable Problem**: For instance, both ice cream sales #of shark attacks may rise during hot weather, but neither causes the other; instead, temperature influences both.

Descriptive Statistics

- Some other functions that are worth exploring:
  - Count()
  - Clip()
  - Rank()
  - Round()
  - https://pandas.pydata.org/pandas-docs/stable/reference/frame.htm

Data Cleansing

Real-world data is messy!

- Missing values
- Outliers in data
- Invalid data (e.g. negative values for ages)
- NaN value (np.nan)
- None value

3 Approaches to enter data

1. Manual
2. Menu
3. Computer generated

Year	City	Amount
1990	New York City	$1,123,456.00
1995-96		2.2 mil
2000s	NYC	No data
2020	New_York	5000000+

https://handsondataviz.org/clean.html

Values considered “missing”

Pandas uses different sentinel values to represent a missing (also referred to as NA) depending on the data type.

- numpy.nan for NumPy data types (original data types will be coerced to np.float64 or object)
- NaT for NumPy np.datetime64, np.timedelta64, and PeriodDtype
- NA for StringDtype, Int64Dtype (and other bit widths), Float64Dtype`(and other bit widths), :class:`BooleanDtype

Use isna() and notna() to detect these missing values

Note: isna() or notna() consider None a missing value

Approaches to manage dirty data

- Replace/impute the value (numeric/categorical)
- Fill gaps forward/backward
- Drop fields
- Interpolation

Year	City	Amount
1990	New York City	$1,123,456.00
1995-96		2.2 mil
2000s	NYC	No data
2020	New_York	5000000+

Replace the values using df.replace()

	0	1
0	-0.349596	-2.017159
1	9999.000000	9999.000000
2	9999.000000	9999.000000
3	0.113889	0.616122
4	0.014707	-1.731660
5	9999.000000	9999.000000
6	1.233087	0.720138
7	9999.000000	9999.000000
8	9999.000000	9999.000000
9	9999.000000	9999.000000

9999.0000

```
df=df.replace(9999.0, 0)
df
```

	0	1
0	-0.349596	-2.017159
1	0.000000	0.000000
2	0.000000	0.000000
3	0.113889	0.616122
4	0.014707	-1.731660
5	0.000000	0.000000
6	1.233087	0.720138
7	0.000000	0.000000
8	0.000000	0.000000
9	0.000000	0.000000

0.0000

Fill missing data gaps with fillna()

replace NA values with non-NA data

```
df
```

	0	1
0	0.061038	1.339673
1	NaN	NaN
2	1.578293	0.637435
3	NaN	NaN
4	NaN	NaN
5	NaN	NaN
6	NaN	NaN
7	NaN	NaN
8	NaN	NaN
9	-1.145787	0.052887

```
df.fillna(method='ffill')
```

	0	1
0	0.061038	1.339673
1	0.061038	1.339673
2	1.578293	0.637435
3	1.578293	0.637435
4	1.578293	0.637435
5	1.578293	0.637435
6	1.578293	0.637435
7	1.578293	0.637435
8	1.578293	0.637435
9	-1.145787	0.052887

```
df.fillna(method='backfill')
```

	0	1
0	0.061038	1.339673
1	1.578293	0.637435
2	1.578293	0.637435
3	-1.145787	0.052887
4	-1.145787	0.052887
5	-1.145787	0.052887
6	-1.145787	0.052887
7	-1.145787	0.052887
8	-1.145787	0.052887
9	-1.145787	0.052887

Forward or backward fill with ffill() and bfill()

fill gaps forward or backward

```
data = {"np": [1.0, np.nan, np.nan, 2], "arrow": pd.array([1.0, pd.NA, pd.NA, 2])}
df = pd.DataFrame(data)
df
```

	np	arrow
0	1.0	1.0
1	NaN	<NA>
2	NaN	<NA>
3	2.0	2.0

Try df.ffill() and df.bfill()?

Drop fields using dropna()

drop rows or columns with missing data

```
df
```

	0	1	2
0	NaN	NaN	-0.335410
1	NaN	NaN	0.685743
2	0.077005	0.085073	0.565144
3	0.394961	-1.829587	0.494039
4	1.486227	-0.480726	-0.127278
5	NaN	NaN	-0.047668
6	NaN	NaN	-1.504804
7	-0.530518	-0.881817	-2.687352
8	0.825376	1.042468	-0.311527
9	0.097617	1.373572	-0.682435

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

	0	1	2
2	0.077005	0.085073	0.565144
3	0.394961	-1.829587	0.494039
4	1.486227	-0.480726	-0.127278
7	-0.530518	-0.881817	-2.687352
8	0.825376	1.042468	-0.311527
9	0.097617	1.373572	-0.682435

May cause data bias if missing values are not MCAR (missing completely at random)

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

	2
0	-0.335410
1	0.685743
2	0.565144
3	0.494039
4	-0.127278
5	-0.047668
6	-1.504804
7	-2.687352
8	-0.311527
9	-0.682435

Drop fields using dropna()

```
dataframe.dropna(axis, how, thresh, subset, inplace)
```

Parameter	Value	Description

axis	`0`
	`1`
	`'index'`
	`'columns'`
		Optional, default 0.
		0 and 'index'removes ROWS that contains NULL values
		1 and 'columns' removes COLUMNS that contains NULL values

how	`'all'` `'any'`	Optional, default 'any'. Specifies whether to remove the row or column when ALL values are NULL, or if ANY value is NULL.
thresh	*Number*	Optional, Specifies the number of NOT NULL values required to keep the row.
subset	*List*	Optional, specifies where to look for NULL values
inplace	`True` `False`	Optional, default False. If True: the removing is done on the current DataFrame. If False: returns a copy where the removing is done.

https://www.w3schools.com/python/pandas/ref_df_dropna.asp

Linear interpolation

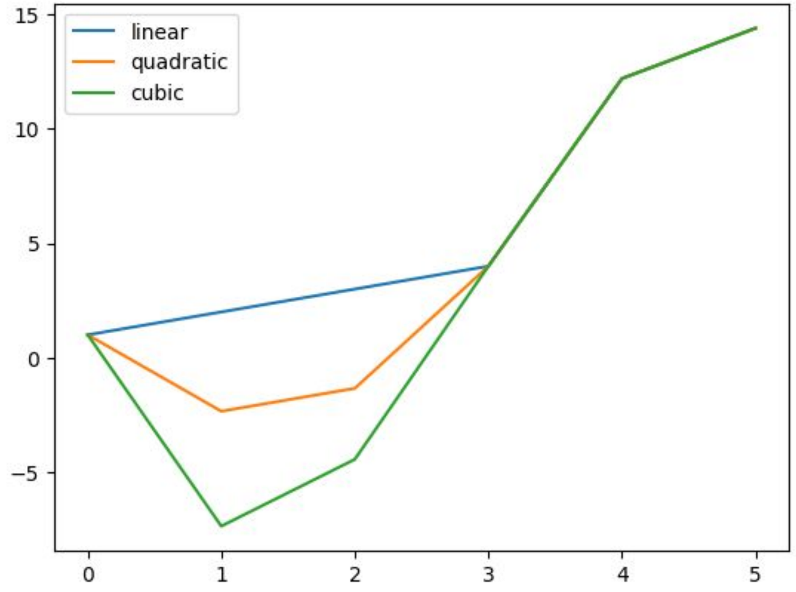
There are many methods to interpolate:

- polynomial interpolation
- regular expressions replacement

```
ser = pd.Series([1, np.nan, np.nan, 4, 12.2, 14.4])
ser
```

	0
0	1.0
1	NaN
2	NaN
3	4.0
4	12.2
5	14.4

```
methods = ["linear", "quadratic", "cubic"]
df = pd.DataFrame({m: ser.interpolate(method=m) for m in methods})
df.plot()
```



Sample Data Wrangling Operations

Slice-out columns

```
df
```

	sensor1	sensor2	sensor3
0	-0.260156	-1.666998	-0.492616
1	-0.762055	-1.114774	0.396817
2	-1.263953	-0.562550	-1.670459
3	-0.765183	-0.619429	0.316981
4	-0.165719	-0.678431	0.485722
5	-1.243191	0.494006	0.145171
6	0.373786	-0.769120	-0.929956
7	-0.081455	0.229613	-2.251816
8	-0.536697	1.228345	-1.040728
9	0.254220	-0.021794	1.268333

```
df['sensor1']
```

```
0   -0.260156
1   -0.762055
2   -1.263953
3   -0.765183
4   -0.165719
5   -1.243191
6    0.373786
7   -0.081455
8   -0.536697
9    0.254220
Name: sensor1, dtype: float64
```

Filter Out Rows

```
df
```

	sensor1	sensor2	sensor3
0	-0.260156	-1.666998	-0.492616
1	-0.762055	-1.114774	0.396817
2	-1.263953	-0.562550	-1.670459
3	-0.765183	-0.619429	0.316981
4	-0.165719	-0.678431	0.485722
5	-1.243191	0.494006	0.145171
6	0.373786	-0.769120	-0.929956
7	-0.081455	0.229613	-2.251816
8	-0.536697	1.228345	-1.040728
9	0.254220	-0.021794	1.268333

```
#Select rows where sensor2 is positive
df[df['sensor2'] > 0]
```

Insert New Column

```
df
```

	sensor1	sensor2	sensor3
0	-0.260156	-1.666998	-0.492616
1	-0.762055	-1.114774	0.396817
2	-1.263953	-0.562550	-1.670459
3	-0.765183	-0.619429	0.316981
4	-0.165719	-0.678431	0.485722
5	-1.243191	0.494006	0.145171
6	0.373786	-0.769120	-0.929956
7	-0.081455	0.229613	-2.251816
8	-0.536697	1.228345	-1.040728
9	0.254220	-0.021794	1.268333

```
df['sensor4']=df['sensor3']**2
```

```
df
```

	sensor1	sensor2	sensor3	sensor4
0	-0.260156	-1.666998	-0.492616	0.242671
1	-0.762055	-1.114774	0.396817	0.157464
2	-1.263953	-0.562550	-1.670459	2.790434
3	-0.765183	-0.619429	0.316981	0.100477
4	-0.165719	-0.678431	0.485722	0.235925
5	-1.243191	0.494006	0.145171	0.021075
6	0.373786	-0.769120	-0.929956	0.864819
7	-0.081455	0.229613	-2.251816	5.070676
8	-0.536697	1.228345	-1.040728	1.083115
9	0.254220	-0.021794	1.268333	1.608669

Add A New Row

```
df
```

	sensor1	sensor2	sensor3	sensor4
0	-0.260156	-1.666998	-0.492616	0.242671
1	-0.762055	-1.114774	0.396817	0.157464
2	-1.263953	-0.562550	-1.670459	2.790434
3	-0.765183	-0.619429	0.316981	0.100477
4	-0.165719	-0.678431	0.485722	0.235925
5	-1.243191	0.494006	0.145171	0.021075
6	0.373786	-0.769120	-0.929956	0.864819
7	-0.081455	0.229613	-2.251816	5.070676
8	-0.536697	1.228345	-1.040728	1.083115
9	0.254220	-0.021794	1.268333	1.608669

```
df.loc[10] = [11,22,33,44]
```

```
df
```

	sensor1	sensor2	sensor3	sensor4
0	-0.260156	-1.666998	-0.492616	0.242671
1	-0.762055	-1.114774	0.396817	0.157464
2	-1.263953	-0.562550	-1.670459	2.790434
3	-0.765183	-0.619429	0.316981	0.100477
4	-0.165719	-0.678431	0.485722	0.235925
5	-1.243191	0.494006	0.145171	0.021075
6	0.373786	-0.769120	-0.929956	0.864819
7	-0.081455	0.229613	-2.251816	5.070676
8	-0.536697	1.228345	-1.040728	1.083115
9	0.254220	-0.021794	1.268333	1.608669
10	11.000000	22.000000	33.000000	44.000000

Delete A Column

```
df
```

	sensor1	sensor2	sensor3
0	-0.260156	-1.666998	-0.492616
1	-0.762055	-1.114774	0.396817
2	-1.263953	-0.562550	-1.670459
3	-0.765183	-0.619429	0.316981
4	-0.165719	-0.678431	0.485722
5	-1.243191	0.494006	0.145171
6	0.373786	-0.769120	-0.929956
7	-0.081455	0.229613	-2.251816
8	-0.536697	1.228345	-1.040728
9	0.254220	-0.021794	1.268333

```
del df['sensor1']
```

```
df
```

	sensor2	sensor3	sensor4
0	-1.666998	-0.492616	0.242671
1	-1.114774	0.396817	0.157464
2	-0.562550	-1.670459	2.790434
3	-0.619429	0.316981	0.100477
4	-0.678431	0.485722	0.235925
5	0.494006	0.145171	0.021075
6	-0.769120	-0.929956	0.864819
7	0.229613	-2.251816	5.070676
8	1.228345	-1.040728	1.083115
9	-0.021794	1.268333	1.608669

Drop specified labels from rows or columns.

```
DataFrame.drop(labels=None, *, axis=0, index=None, columns=None,
level=None, inplace=False, errors='raise')
```

[source]

```
df = pd.DataFrame(
  [[88, 72, 67],
  [23, 78, 62],
  [55, 54, 76]],
  columns=['a', 'b', 'c'])
print(df)
```

```
    a   b   c
0  88  72  67
1  23  78  62
2  55  54  76
```

```
df.drop(index = [1,2])
```

```
df.drop(columns = ['a'])
```

Groupby and Concatenation

To create a structure

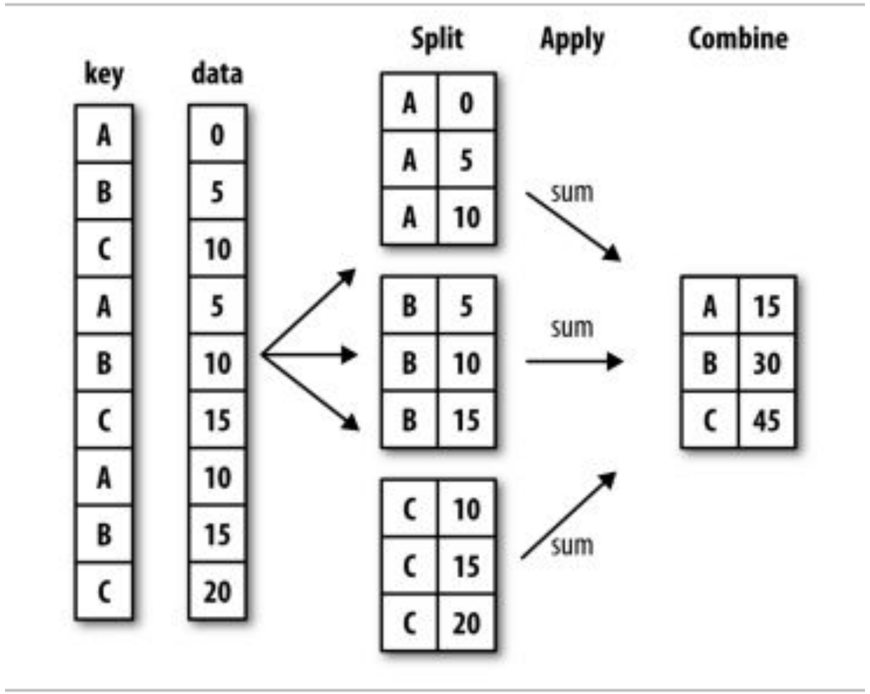
Data aggregation and Group Operation

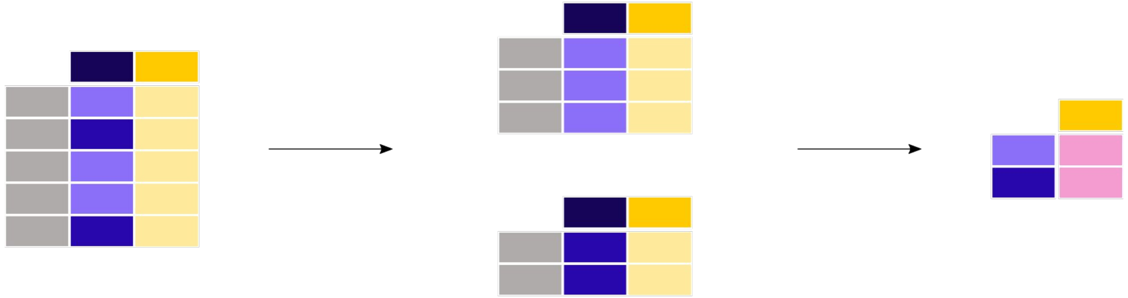
- Categorizing a dataset and applying a function to each group is often a critical component of data analysis workflow
  - Compute group statistics (sum, mean, count)
  - Pivot tables for reporting or visualization purposes
- Pandas provides a flexible **groupby** interface, enabling users to **split**, **apply** and **combine** datasets in a natural way.
- Grouby: a processing involving one or more of the following steps:
  - **Splitting** the data into groups based on some criteria
  - **Applying** a function to each group independently (Aggregation, Transformation, Filtration
  - **Combining** results into a data structure

```
SELECT Column1, Column2, mean(Column3), sum(Column4)
FROM SomeTable
GROUP BY Column1, Column2
```

GroupBy and Aggregate (Split-Apply-Combine)

- **Splitting** the data into groups based on some criteria
- **Applying** a function to each group independently
  - **Aggregation**: compute sum, mean, count
  - **Transformation**: standardized data, impute NA with some specified value
  - **Filtration**: discard some groups
- **Combining** results into a data structure





Splitting an object into groups

```
np.random.seed(12345)
df = pd.DataFrame({'key1' : ['a', 'a', 'b', 'b', 'a'],
                   'key2' : ['one', 'two', 'one', 'two', 'one'],
                   'data1' : np.random.randint(0,10,5),
                   'data2' : np.random.randint(0,10,5)})
df
```

	key1	key2	data1	data2
0	a	one	2	5
1	a	two	5	2
2	b	one	1	1
3	b	two	4	6
4	a	one	9	1

```
group1 = df.groupby('key1')
group2 = df.groupby(['key1','key2'])
```

- **Grouping**: A mapping of labels to group names.
- The mapping can be specified many different ways:
  - For DataFrame objects, a string indicating either a column name or an index level name to be used to group.
  - A Python function, to be called on each of the index labels.
  - A list or NumPy array of the same length as the index.
  - A dict or Series, providing a label -> group name mapping.
  - A list of any of the above things.
- Collectively we refer to the grouping objects as the **keys**.

Splitting an object into groups: aggregation

A GroupBy operation that reduces the dimension of the grouping object.

```
np.random.seed(12345)
df = pd.DataFrame({'key1' : ['a', 'a', 'b', 'b', 'a'],
                   'key2' : ['one', 'two', 'one', 'two', 'one'],
                   'data1' : np.random.randint(0,10,5),
                   'data2' : np.random.randint(0,10,5)})
df
```

	key1	key2	data1	data2
0	a	one	2	5
1	a	two	5	2
2	b	one	1	1
3	b	two	4	6
4	a	one	9	1

```
group1 = df.groupby('key1')
group2 = df.groupby(['key1','key2'])
```

```
group1.sum()
```

key1	key2	data1	data2
a	onetwoone	16	8
b	onetwo	5	7

sorted by default

```
group2.sum()
```

key1	key2	data1	data2
a	**one**		
	two		
b	**one**		
	two		

Sample built-in aggregation methods

Method	Description
any()	Compute whether any of the values in the groups are truthy
all()	Compute whether all of the values in the groups are truthy
count()	Compute the number of non-NA values in the groups
cov() *	Compute the covariance of the groups
first()	Compute the first occurring value in each group
idxmax()	Compute the index of the maximum value in each group
idxmin()	Compute the index of the minimum value in each group
last()	Compute the last occurring value in each group

The `aggregate()` method can accept many different types of inputs.

- Any reduction method that pandas implements can be passed as a string
- User-defined function (lambda style)

Splitting an object into groups: transformation

A groupby operation whose result is indexed as the one being grouped.

```
np.random.seed(12345)
df = pd.DataFrame({'key1' : ['a', 'a', 'b', 'b', 'a'],
                   'key2' : ['one', 'two', 'one', 'two', 'one'],
                   'data1' : np.random.randint(0,10,5),
                   'data2' : np.random.randint(0,10,5)})
df
```

	key1	key2	data1	data2
0	a	one	2	5
1	a	two	5	2
2	b	one	1	1
3	b	two	4	6
4	a	one	9	1

```
group1 = df.groupby('key1')[['data1','data2']]
group1.cumsum()
```

	data1	data2
0	2	5
1	7	7
2	1	1
3	5	7
4	16	8

Sample built-in transformation methods

Method	Description
`bfill()`	Back fill NA values within each group
`cumcount()`	Compute the cumulative count within each group
`cummax()`	Compute the cumulative max within each group
`cummin()`	Compute the cumulative min within each group
`cumprod()`	Compute the cumulative product within each group
`cumsum()`	Compute the cumulative sum within each group
`diff()`	Compute the difference between adjacent values within each group
`ffill()`	Forward fill NA values within each group

The `transform()` method can accept many different types of inputs (similar to aggregate).

- Any reduction method that pandas implements can be passed as a string
- User-defined function (lambda style)

Sample built-in filtration methods

Method	Description
head()	Select the top row(s) of each group
nth()	Select the nth row(s) of each group
tail()	Select the bottom row(s) of each group

Combining and Merging Datasets

- Merging and joining tables or datasets are highly common operations for a data wrangling professional.
- Data contained in pandas objects can be combined together in a number of ways
- pandas.merge connects rows in DataFrames based on one or more keys.
- pandas.concat concatenates or “stacks” together objects along an
- axis.

Concatenating Along an Axis

- pandas.concat: concatenate pandas objects **along a particular axis** with optional set logic along the other axes.
- By default, the concat() function works on axis = 0 (index or row)

```
df1 = pd.DataFrame([['a', 1], ['b', 2]],
                   columns=['letter', 'number'])
df3 = pd.DataFrame([['c', 3, 'cat'], ['d', 4, 'dog']],
                   columns=['letter', 'number', 'animal'])
```

df1

	letter	number
0	a	1
1	b	2

df3

	letter	number	animal
0	c	3	cat
1	d	4	dog

```
pd.concat([df1, df3])
```

	letter	number	animal
0	a	1	NaN
1	b	2	NaN
0	c	3	cat
1	d	4	dog

```
pd.concat([df1,df3],axis=1)
```

	letter	number	letter	number	animal
0	a	1	c	3	cat
1	b	2	d	4	dog

pandas.concat

- The problem with this kind of operation is that the **concatenated parts are not identifiable in the result**.
- Use the **keys option to create a hierarchical index** on the axis of concatenation.

```
pd.concat([df1, df3], keys=['df1','df3'])
```

		letter	number	animal
df1	0	a	1	NaN
	1	b	2	NaN
df3	0	c	3	cat
	1	d	4	dog

```
pd.concat([df1, df3], keys=['df1','df3'], axis=1)
```

	df1		df3		
	letter	number	letter	number	animal
0	a	1	c	3	cat
1	b	2	d	4	dog

Database-Style DataFrame Joins

- *Merge* or *join* operations combine datasets by linking rows using one or more *keys*.

```
pd.merge(df1, df2, on='key', how=‘inner')
```

- *Different join types with how argument*
  - `'inner'` Use only the key combinations observed in both tables (default)
  - `'left'` Use all key combinations found in the left table
  - `'right'` Use all key combinations found in the right table
  - `'outer'` Use all key combinations observed in both tables together

pandas.DataFrame.merge

df1

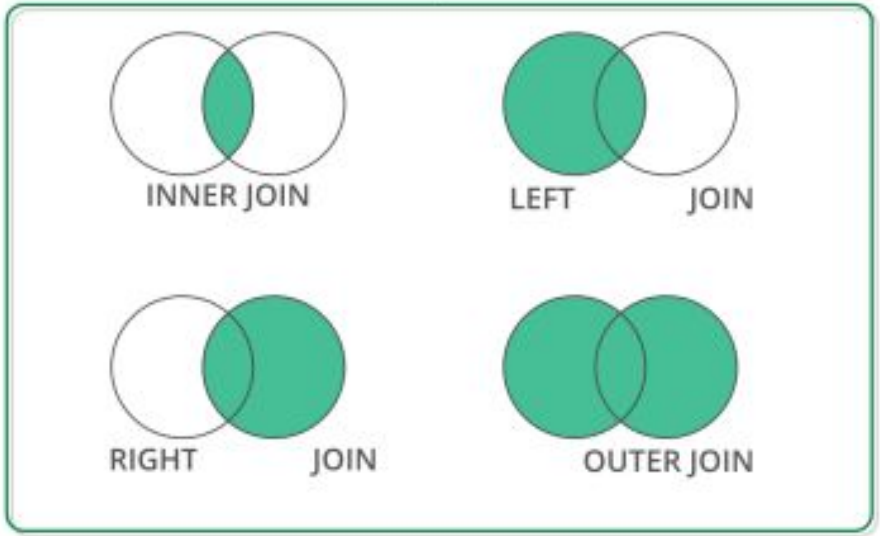
	lkey	data1
0	b	0
1	b	1
2	a	2
3	c	3
4	a	4
5	a	5
6	b	6

df2

	rkey	data2
0	a	0
1	b	1
2	d	2

```
df1 = pd.DataFrame({'lkey': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],'data1': range(7)})
df2 = pd.DataFrame({'rkey': ['a', 'b', 'd'],'data2': range(3)})
pd.merge(df1, df2, left_on='lkey', right_on='rkey', how='inner')
```

	lkey	data1	rkey	data2
0	b	0	b	1
1	b	1	b	1
2	b	6	b	1
3	a	2	a	0
4	a	4	a	0
5	a	5	a	0



Stack DataFrames using append()

```
left.append(right)
```

	_key1	_key2	city	hire_date	profession	user_name
0	K0	z0	city_0	NaN	NaN	user_0
1	K1	z1	city_1	NaN	NaN	user_1
2	K2	z2	city_2	NaN	NaN	user_2
3	K3	z3	city_3	NaN	NaN	user_3
0	K0	z0	NaN	h_0	p_0	NaN
1	K1	z1	NaN	h_1	p_1	NaN
2	K2	z2	NaN	h_2	p_2	NaN
3	K3	z3	NaN	h_3	p_3	NaN

Append to itself (a dataframe) not creating a new one

String manipulation

To deal with string data type

Working with text data

There are two ways to store text data in pandas:

- object dtype NumPy array
- StringDtype extension type (recommended)

```
In [1]: pd.Series(["a", "b", "c"])
Out[1]:
0    a
1    b
2    c
dtype: object
```

```
In [2]: pd.Series(["a", "b", "c"], dtype="string")
Out[2]:
0    a
1    b
2    c
dtype: string
```

String methods

- Series and Index are equipped with a set of string processing methods that make it easy to operate on each element of the array
- Exclude missing/NA values automatically
- These are accessed via the **str** attribute and generally have names matching the equivalent (scalar) built-in string methods

Sample string methods

- str.lower(), str.upper()
- str.split()
- str.contains()
- str.replace()
- str.extract()

str.split()

In the default setting, the string is split by whitespace

```
df
```

	_key1	_key2	city	user_name	hire_date	profession
0	K0	z0	city_0	user_0	h_0	p_0
1	K1	z1	city_1	user_1	h_1	p_1
2	K2	z2	city_2	user_2	h_2	p_2
3	K3	z3	city_3	user_3	h_3	p_3

```
df['city'].str.split('_')
```

```
0    [city, 0]
1    [city, 1]
2    [city, 2]
3    [city, 3]
dtype: object
```

str.contains()

```
df
```

	_key1	_key2	city	user_name	hire_date	profession
0	K0	z0	city_0	user_0	h_0	p_0
1	K1	z1	city_1	user_1	h_1	p_1
2	K2	z2	city_2	user_2	h_2	p_2
3	K3	z3	city_3	user_3	h_3	p_3

```
df['city'].str.contains('2')
```

```
0    False
1    False
2     True
3    False
Name: city, dtype: bool
```

str.replace()

```
df
```

	_key1	_key2	city	user_name	hire_date	profession
0	K0	z0	city_0	user_0	h_0	p_0
1	K1	z1	city_1	user_1	h_1	p_1
2	K2	z2	city_2	user_2	h_2	p_2
3	K3	z3	city_3	user_3	h_3	p_3

```
df['city'].str.replace('_', '##')
```

```
0    city##0
1    city##1
2    city##2
3    city##3
Name: city, dtype: object
```

str.extract() – Returns first match found (reg*)

- The extract method accepts a regular expression with at least one capture group.
- Elements that do not match return a row filled with NaN.
- A Series of messy strings can be "converted" into a like-indexed Series or DataFrame of cleaned-up or more useful strings

```
x = pd.Series(["a1", "b2", "c3"], dtype="string")
x
```

	0
0	a1
1	b2
2	c3

dtype: string

```
x.str.extract("([ab])")
```

	0
0	a
1	b
2	<NA>

```
x.str.extract("(\d)")
```

	0
0	1
1	2
2	3

Date/time

To deal with date/time data type

Time series / date functionality

Pandas captures 4 general time related concepts:

1. **Date times**: A specific date and time with timezone support. Similar to datetime.datetime from the standard library.
2. **Time spans**: A span of time defined by a point in time and its associated frequency.
3. **Time deltas**: An absolute time duration. Similar to datetime.timedelta from the standard library.
4. **Date offsets**: A relative time duration that respects calendar arithmetic. Similar to dateutil.relativedelta.relativedelta from the dateutil package.

4 general time related concepts:

Concept	Scalar Class	Array Class	pandas Data Type	Primary Creation Method
Date times	`Timestamp`	`DatetimeIndex`	`datetime64[ns]` or `datetime64[ns, tz]`	`to_datetime` or `date_range`
Time deltas	`Timedelta`	`TimedeltaIndex`	`timedelta64[ns]`	`to_timedelta` or `timedelta_range`
Time spans	`Period`	`PeriodIndex`	`period[freq]`	`Period` or `period_range`
Date offsets	`DateOffset`	`None`	`None`	`DateOffset`

Timestamps vs. time spans

- Timestamped data is the most basic type of time series data that associates values with points in time. For pandas objects it means using the points in time.
- TIme span represents the interval.

```
import datetime
t_instant = pd.Timestamp(datetime.datetime(2024, 5, 1,13,10))
t_instant
```

```
Timestamp('2024-05-01 13:10:00')
```

```
t_period = pd.Period("2024-05", freq="D")
t_period
```

```
Period('2024-05-01', 'D')
```

```
print(t_period.start_time, t_period.end_time)
```

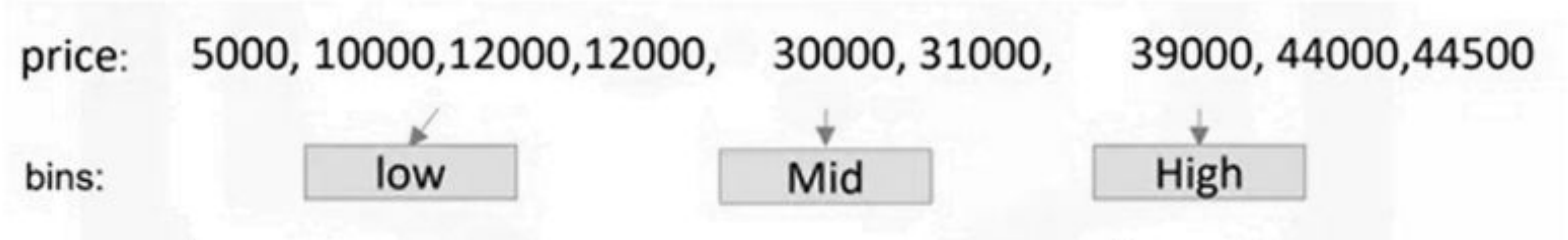
```
2024-05-01 00:00:00 2024-05-01 23:59:59.999999999
```

Discretization and Binning

To turn numeric into categorical

Discretization and Binning

- Continuous data is often discretized or otherwise separated into ‘bins’ for analysis.
- Binning: Grouping of values into bins.
- Convert numeric to categorical variables.
- Group a set of numeric values into a set of bins.
- Similar to VLOOKUP with approximate_match
- Ex: ‘price’ is a feature ranges from 5,000 to 45,000.



Binning in Python pandas

	price
0	5000
1	450000
2	244696
3	141021
4	58224

	price	price_bins
0	5000	Low
1	450000	High
2	244696	Medium
3	141021	Low
4	58224	Low

```
bins = np.linspace(min(df['price']),max(df['price']),4)
group_names=['Low','Medium','High']
df['price_bins'] = pd.cut(df['price'],bins,labels=group_names, include_lowest=True)
```

Binning in Python pandas

Bin counts for the result of df['price_bin']

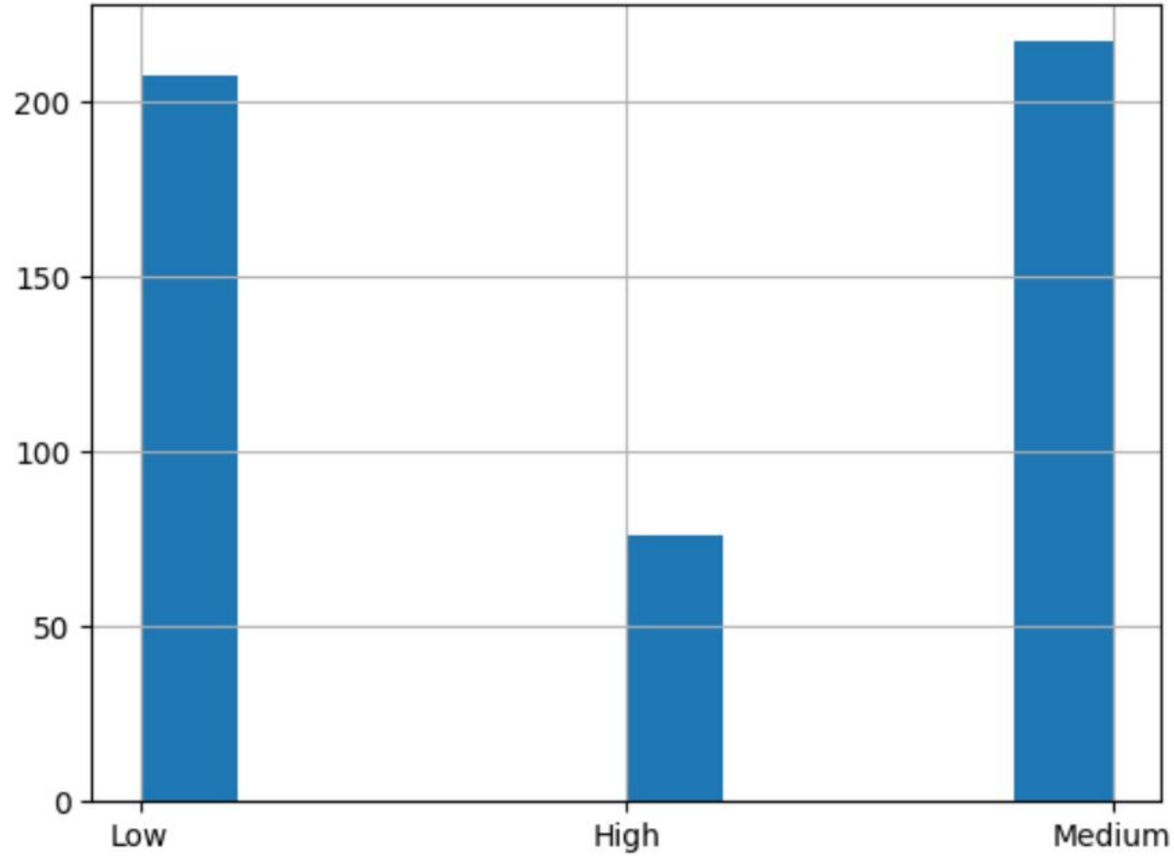
```
df['price_bins'].value_counts()
```

price_bins	count
Medium	217
Low	207
High	76

dtype: int64

```
df['price_bins'].hist()
```

<Axes: >



Turning categorical variables into quantitative variables

- Problem
  - Many machine learning algorithms cannot operate on label data (object/string (categorical variable)) directly. They require all input variables and output variables to be numeric.

	Car	Fuel
0	A	diesel
1	B	gas
2	C	gas
3	D	diesel
4	E	diesel

Turning categorical variables into quantitative variables

Solution: **(One-Hot Encoding)**

- Add dummy variables for each unique category
- Assign 0 or 1 in each category

	Car	Fuel
0	A	diesel
1	B	gas
2	C	gas
3	D	diesel
4	E	diesel

	Car	Fuel	diesel	gas
0	A	diesel	1	0
1	B	gas	0	1
2	C	gas	0	1
3	D	diesel	1	0
4	E	diesel	1	0

The binary variables are often called “dummy variables”

Dummy variables in Python Pandas

- Use pandas.get_dummies() method
- Convert categorical variables into dummy variables (0 or 1)

```
df_dummy=pd.get_dummies(df['Fuel'], dtype='int')
df_dummy
```

	diesel	gas
0	1	0
1	0	1
2	1	0
3	1	0
4	0	1

```
df_with_dummy = df.join(df_dummy)
df_with_dummy
```

	Car	Fuel	diesel	gas
0	A	diesel	1	0
1	B	gas	0	1
2	C	diesel	1	0
3	D	diesel	1	0
4	E	gas	0	1

References

- https://pandas.pydata.org/docs/index.html
- Harvard University: CS109A Data Science
- edX: Python for Data Science, UCSanDiegoX (DSE200x).
- Coursera: Introduction to Python for Data Science, Microsoft (DAT208x)
- https://aws.amazon.com/blogs/machine-learning/integrate-amazon-sagemaker-data-wrangler-with-mlops-workflows/
- https://docs.python.org/3/howto/regex.html#regex-howto