

Data Tidying and Cleaning

What's the most efficient way to perform data transformations in pandas: Step-by-Step guide

Step 1: Import the necessary libraries

The first step is to import the necessary libraries. In this case, you will need to import pandas. You can do this by typing the following command in your Python environment:

```
import pandas as pd
```

Step 2: Load your data

The next step is to load your data into a pandas DataFrame. You can do this using the pandas `read_csv()` function if your data is in a CSV file. For example:

```
df = pd.read_csv('your_file.csv')
```

Step 3: Inspect your data

Before you start transforming your data, it's a good idea to inspect it first to understand its structure and content. You can do this using the `head()` function, which returns the first `n` rows of your DataFrame. For example:

```
df.head()
```

Step 4: Perform data transformations

There are many ways to perform data transformations in pandas, but the most efficient way is usually to use vectorized operations. These are operations that are performed on entire arrays of data at once, rather than on individual elements. This can significantly speed up your data processing.

Here are a few examples of common data transformations and how to perform them in a vectorized way:

Adding a new column based on existing columns:

```
df['new_column'] = df['column1'] + df['column2']
```

Applying a function to a column:

```
df['column'] = df['column'].apply(lambda x: x**2)
```

Replacing values in a column:

```
df['column'] = df['column'].replace({'old_value': 'new_value'})
```

Step 5: Check your transformations

After performing your transformations, it's a good idea to check that they have been applied correctly. You can do this by inspecting your DataFrame again using the `head()` function.

Step 6: Save your transformed data

Finally, once you're happy with your transformations, you can save your transformed data back to a CSV file using the `to_csv()` function. For example:

```
df.to_csv('your_transformed_file.csv', index=False)
```

Remember, the key to efficient data transformations in pandas is to use vectorized operations wherever possible. This will ensure that your transformations are performed as quickly and efficiently as possible.

Melt

```
] : tb_tidy = tb.melt(id_vars = ["iso2", "year"], var_name = "sex_and_age", value_name = "cases")
```

```
] : tb_tidy.head()
```

```
] :
```

	iso2	year	sex_and_age	cases
0	AD	1989	m04	NaN
1	AD	1990	m04	NaN
2	AD	1991	m04	NaN
3	AD	1992	m04	NaN
4	AD	1993	m04	NaN

Slice

```
1]: tb_tidy["sex"] = tb_tidy.sex_and_age.str.slice(0, 1)
```

```
1]:
```

0	m
1	m
2	m
3	m
4	m
..	
115375	f
115376	f
115377	f
115378	f
115379	f

Name: sex_and_age, Length: 115380, dtype: object

```
0]: tb_tidy.sex_and_age.str.slice(1).unique()
```

```
0]: array(['04', '514', '014', '1524', '2534', '3544', '4554', '5564', '65',  
        'u'], dtype=object)
```

Slices

```
] : tb_tidy["gender"] = tb_tidy.sex_and_age.str.slice(0, 1)
```

```
] : tb_tidy["age_group"] = tb_tidy.sex_and_age.str.slice(1)
```

```
] : tb_tidy
```

```
] :
```

	iso2	year	sex_and_age	cases	gender	age_group
0	AD	1989	m04	NaN	m	04
1	AD	1990	m04	NaN	m	04
2	AD	1991	m04	NaN	m	04
3	AD	1992	m04	NaN	m	04
4	AD	1993	m04	NaN	m	04
...
115375	ZW	2004	fu	NaN	f	u
115376	ZW	2005	fu	NaN	f	u
115377	ZW	2006	fu	NaN	f	u
115378	ZW	2007	fu	NaN	f	u
115379	ZW	2008	fu	0.0	f	u

Drop column

```
tb_tidy = tb_tidy.drop(columns = ["sex_and_age"])
```

Mean or whatever by condition

```
: tb_tidy_no_missing[tb_tidy_no_missing.iso2 == "BG"].cases.mean()
```

Values count

```
tb_tidy_no_missing.age_group.str.len().value_counts()
```

Apply and function

```
5]: tb_tidy_no_missing.age_group.apply(lambda x: x[0])
```

```
5]: 15      0
    16      0
    18      0
    42      0
    43      0
    ..
    115195   u
    115269   u
    115323   u
    115350   u
    115379   u
    Name: age_group, Length: 35552, dtype: object
```

```
] : def process_age_group(age_group):
    ages = {"04": "0-4", "65": "65+", "u": "unknown"}
    if age_group in ages:
        return ages[age_group]
    else:
        # Put a dash before the last two digits
        return f"{age_group[:-2]}-{age_group[-2:]}"
```

Apply function

```
def process_age_group(age_group):
    ages = {"04": "0-4", "65": "65+", "u": "unknown"}
    if age_group in ages:
        return ages[age_group]
    # Put a dash before the last two digits
    return f"{age_group[:-2]}-{age_group[-2:]}"
```

```
tb_tidy_no_missing.age_group.apply(process_age_group)
```

Split and expand

```
tb_tidy_no_missing.age_group.str.split("-", expand = True)
```

	0	1
15	0	4
16	0	4

To category

```
tb_tidy_no_missing.gender = tb_tidy_no_missing.gender.astype("category")
tb_tidy_no_missing.age_group = tb_tidy_no_missing.age_group.astype("category")
```

Rearrange columns

```
tb_tidy_no_missing[["iso2", "year", "gender", "age_group", "cases"]]
```

Sort by iso2 and then by year

```
tb_tidy_no_missing.sort_values(["iso2", "year"])
```

```
tb_tidy_no_missing = tb_tidy_no_missing.sort_values(["iso2", "year"])
```

Reset index - rearrange

```
tb_tidy_no_missing.reset_index()
```

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

Save to another csv

```
tb_tidy_no_missing.to_csv("data/tb_tidy.csv", index = None)
```

Loc - save new value "d31" on row 2, 23 column

```
weather_data.loc[2, "d31"] = 23
```

Melt, slice, dropna

```
weather_data = pd.read_csv("data/weather.csv")
```

```
weather_data_tidy = weather_data.melt(id_vars = ["id", "year", "month", "element"], var_name = "day")
```

```
weather_data_tidy.day = weather_data_tidy.day.str.slice(1).astype(int)
```

```
weather_data_tidy = weather_data_tidy.dropna()
```

```
weather_data
```

Pivot

```
weather_data_tidy.pivot_table(columns = "element", values = "value")
```

element	tmax	tmin
value	29.190909	14.651515

```
|: weather_data_tidy.pivot_table(index = ["id", "year", "month"], columns = "element", values = "value")
```

```
|:
```

		element	tmax	tmin	
	id	year	month		
	MX17004	2010	1	27.800000	14.500000
			2	27.750000	13.225000
			3	32.566667	16.200000

```
weather_data_tidy = weather_data_tidy.pivot_table(index = ["id", "year", "month"], columns = "element", values = "value")
```

```
weather_data_tidy.reset_index()
```

Which are in week 50? - notna

```
billboard_data[billboard_data.wk50.notna()]
```

Last 15 columns

```
billboard_data.columns[-15:]
Index(['wk62', 'wk63', 'wk64', 'wk65', 'wk66', 'wk67', 'wk68', 'wk69', 'wk70',
      'wk71', 'wk72', 'wk73', 'wk74', 'wk75', 'wk76'],
      dtype='object')
```

Melt more examples

```
billboard_data = billboard_data.melt(
    id_vars = ["year", "artist", "track", "time", "date.entered"],
    var_name = "week",
    value_name = "position"
)
```

Slice and astype

```
billboard_data.week = billboard_data.week.str.slice(2).astype(int)
```

Data datetime

```
pd.to_datetime(billboard_data["date.entered"])
```

```
billboard_data["date.entered"] = pd.to_datetime(billboard_data["date.entered"])
```

Add week of the year

```
billboard_data["date.entered"].dt.add()

pd.Timedelta(1, "w")

Timedelta('7 days 00:00:00')

intervals = billboard_data.week.apply(lambda x: pd.Timedelta(x, "w"))

billboard_data["date"] = billboard_data["date.entered"] + intervals

C:\Users\Yordan\AppData\Local\Temp\ipykernel_20440\532179450.py:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-d
billboard_data["date"] = billboard_data["date.entered"] + intervals

billboard_data
```



```
Timedelta('7 days 00:00:00')
```

```
intervals = billboard_data.week.apply(lambda x: pd.Timedelta(x - 1, "w"))
```

```
billboard_data["date"] = billboard_data["date.entered"] + intervals
```

```
billboard_data
```

	year	artist	track	time	date.entered	week	position	date
0	2000	2 Pac	Baby Don't Cry (Keep...	4:22	2000-02-26	1	87	2000-03-04
1	2000	2Ge+her	The Hardest Part Of ...	3:15	2000-09-02	1	91	2000-09-09
2	2000	3 Doors Down	Kryptonite	3:53	2000-04-08	1	81	2000-04-15

Drop and rearrange columns

```
billboard_data = billboard_data.drop(columns = ["date.entered", "week"])
```

```
billboard_data = billboard_data[["year", "artist", "track", "time", "date", "position"]]
```

```
billboard_data
```

	year	artist	track	time	date	position
0	2000	2 Pac	Baby Don't Cry (Keep...	4:22	2000-02-26	87
1	2000	2Ge+her	The Hardest Part Of ...	3:15	2000-09-02	91
2	2000	3 Doors Down	Kryptonite	3:53	2000-04-08	81

Group by

```
billboard_data.groupby(["track", "artist"]).value_counts()
```

```
track          artist      year  time  date      position
(Hot S**t) Country G...  Nelly    2000  4:17  2000-04-29    100      1
                                     2000-05-06    99      1
                                     2000-09-02    11      1
                                     2000-08-26    11      1
                                     2000-08-19    15      1
...
www.memory            Jackson, Alan  2000  2:36  2000-11-25    54      1
                                     2000-11-18    59      1
-----
```

Condition if

```
billboard_data[billboard_data.position == 1].artist.unique()
```

```
array(['Madonna', 'Aguilera, Christina', 'Sisqo', 'Destiny's Child',
       'Santana', 'Carey, Mariah', 'Janet', 'Iglesias, Enrique', 'N'Sync',
       'matchbox twenty', 'Aaliyah', 'Savage Garden', 'Vertical Horizon',
       'Creed', 'Lonestar'], dtype=object)
```

```
billboard_data[billboard_data.position == 1].artist.value_counts()
```

```
artist
Destiny's Child    14
Santana            10
Aguilera, Christina  6
Madonna            4
```



Read csv from link - processing, basic steps

```
weather_data = pd.read_csv("https://raw.githubusercontent.com/synesthesiam/blog/master/posts/data/weather_year.csv")
```

```
weather_data
```

```
weather_data.columns = ["date", "max_temp", "mean_temp", "min_temp", "max_dew", "mean_dew", "min_dew", "max_humidity", "mean_humidity",  
"min_humidity", "max_pressure", "mean_pressure", "min_pressure", "max_visibility", "mean_visibility", "min_visibility", "max_wind",  
"mean_wind", "max_gusts", "precipitation", "cloud_cover", "events", "wind_dir"]
```

Convert column names

```
weather_data.columns
```

```
Index(['EDT', 'Max TemperatureF', 'Mean TemperatureF', 'Min TemperatureF',  
      'Max Dew PointF', 'MeanDew PointF', 'Min DewpointF', 'Max Humidity',  
      ' Mean Humidity', ' Min Humidity', ' Max Sea Level PressureIn',  
      ' Mean Sea Level PressureIn', ' Min Sea Level PressureIn',  
      ' Max VisibilityMiles', ' Mean VisibilityMiles', ' Min VisibilityMiles',  
      ' Max Wind SpeedMPH', ' Mean Wind SpeedMPH', ' Max Gust SpeedMPH',  
      'PrecipitationIn', ' CloudCover', ' Events', ' WindDirDegrees'],  
      dtype='object')
```

```
weather_data.columns = ["date", "max_temp", "mean_temp", "min_temp", "max_dew", "mean_dew", "min_dew", "max_humidity", "mean_humidity",  
"min_humidity", "max_pressure", "mean_pressure", "min_pressure", "max_visibility", "mean_visibility", "min_visibility", "max_wind",  
"mean_wind", "max_gusts", "precipitation", "cloud_cover", "events", "wind_dir"]
```

To datetime

```
pd.to_datetime(weather_data.date)
```

```
0    2012-03-10  
1    2012-03-11
```

Dtypes

```
weather_data.dtypes
```

```
date          datetime64[ns]  
max_temp      int64  
mean_temp     int64
```

Object column analysis

```
weather_data.precipitation.unique()
```

```
array(['0.00', 'T', '0.03', '0.04', '0.14', '0.86', '0.06', '0.01',  
      '0.51', '0.69', '1.45', '0.38', '0.19', '0.15', '0.49', '0.29',  
      '0.09', '0.90', '0.02', '0.07', '0.13', '0.10', '0.36', '0.27',  
      '0.16', '0.26', '0.31', '0.05', '0.32', '1.85', '0.53', '2.00',  
      '0.92', '1.10', '0.17', '1.13', '0.63', '0.50', '0.71', '0.73',  
      '1.52', '0.47', '0.39', '0.18', '0.77', '0.08', '0.33', '0.44',  
      '0.48', '0.20', '0.12', '0.82', '1.16', '1.73', '0.40', '0.99',  
      '0.30', '1.17'], dtype=object)
```

```
weather_data.precipitation == "T"
```

```
0    False  
1     True
```

How many are with 'T'

```
weather_data[weather_data.precipitation == "T"]
```

Replace column value 'T' with a very small number

```
weather_data.loc[weather_data.precipitation == "T", "precipitation"] = 1e-6
```

Astype float

```
weather_data.precipitation.astype(float)
```

```
0    0.000000
1    0.000001
2    0.030000
3    0.000000
4    0.000000
...
```

```
weather_data.precipitation = weather_data.precipitation.astype(float)
```

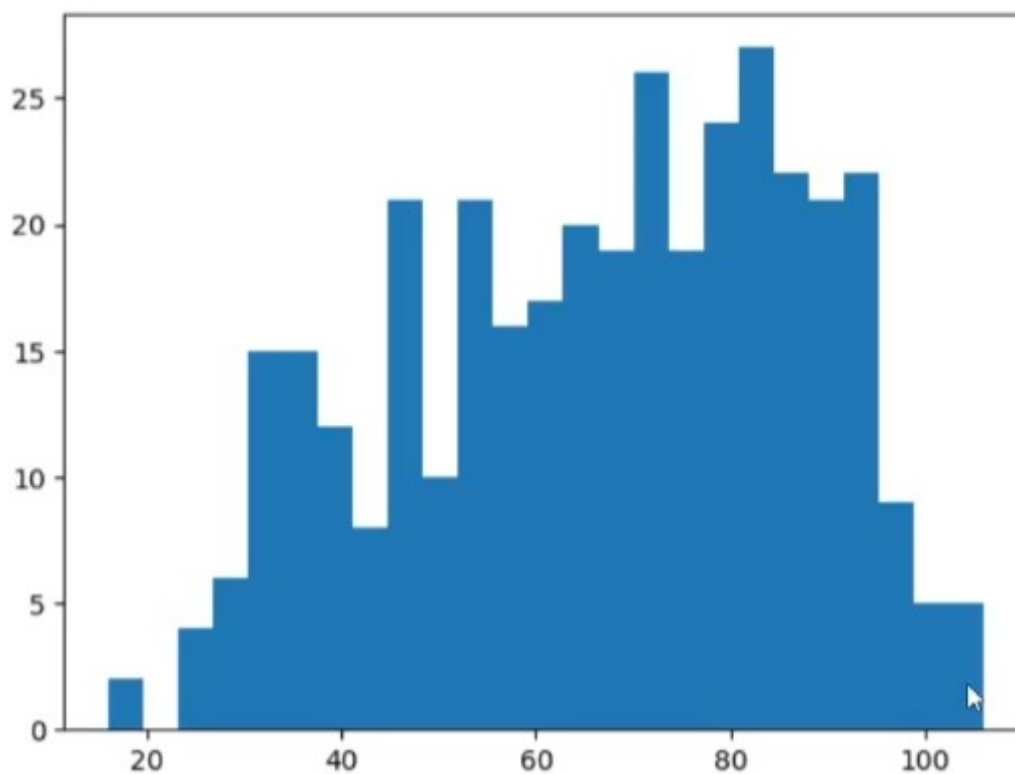
Count events including NAN

```
weather_data.events.value_counts(dropna = False)
```

```
events
NaN                204
Rain               69
Rain-Thunderstorm  26
Fog               13
Snow             13
Fog-Rain         11
Thunderstorm      9
Fog-Rain-Thunderstorm  8
_                -
```

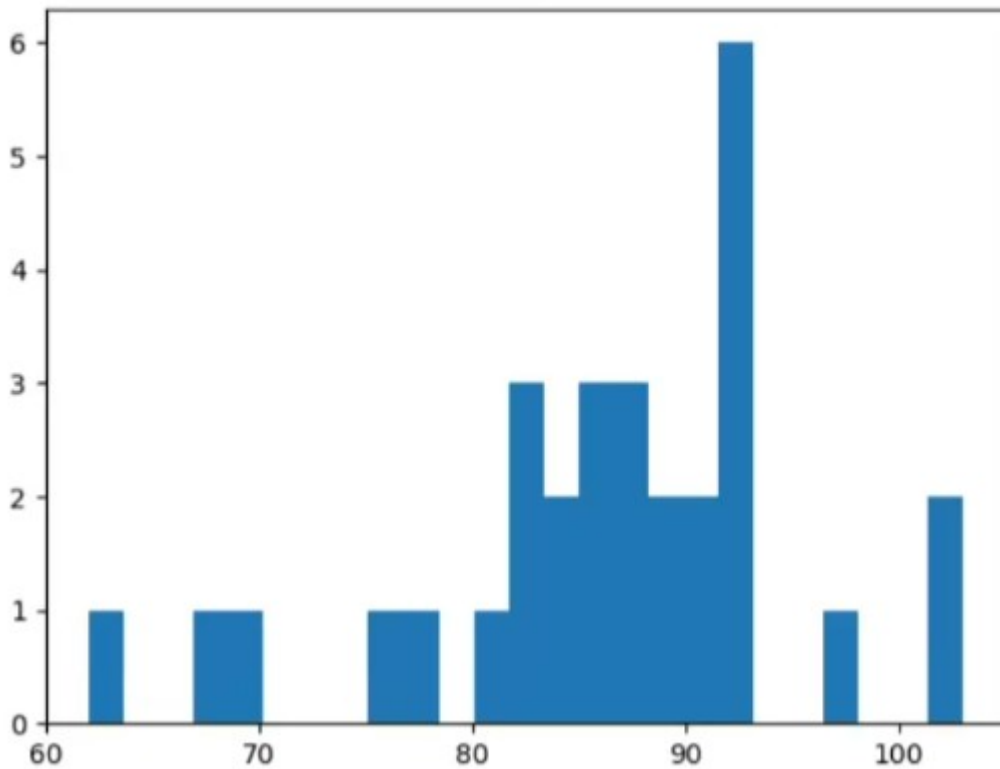
Hist

```
plt.hist(weather_data.max_temp, bins = 25)
plt.show()
```



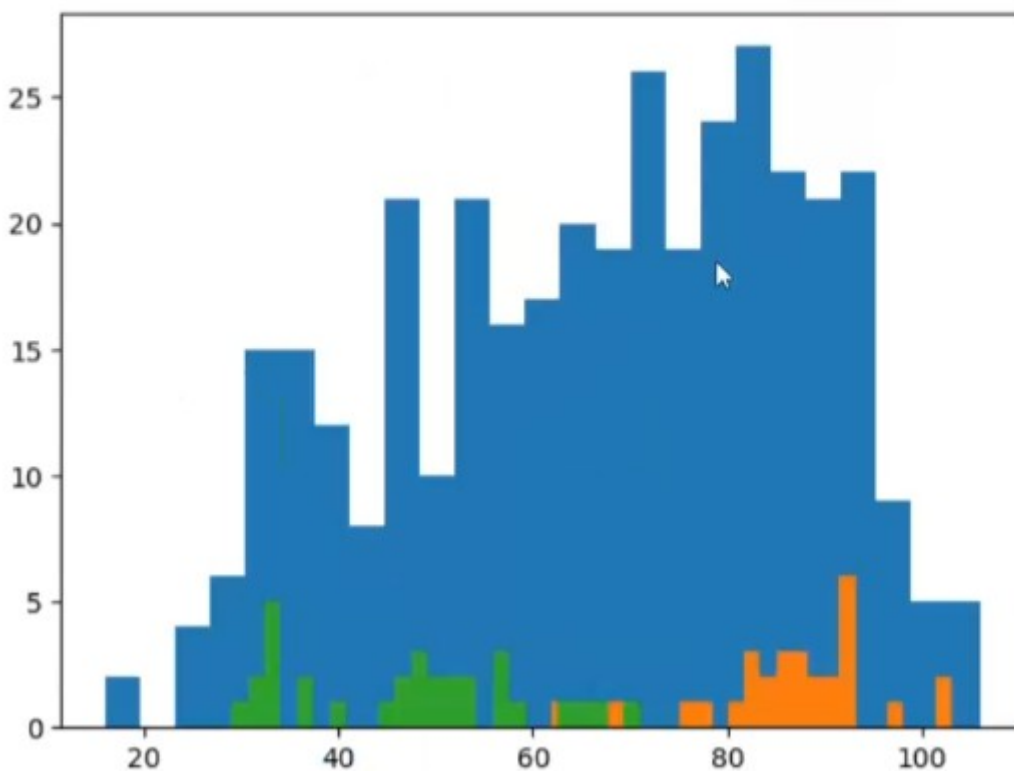
Hist max temperatures for june - 6

```
plt.hist(weather_data[weather_data.date.dt.month == 6].max_temp, bins = 25)  
plt.show()
```



Plot all together

```
plt.hist(weather_data.max_temp, bins = 25)  
plt.hist(weather_data[weather_data.date.dt.month == 6].max_temp, bins = 25)  
plt.hist(weather_data[weather_data.date.dt.month == 12].max_temp, bins = 25)  
  
plt.show()
```



Average temperature for every week

```
weather_data = weather_data.set_index("date")
```

```
weather_data.resample("w").min_temp.mean()
```

```
date
2012-03-11    27.000000
2012-03-18    51.714286
2012-03-25    54.857143
2012-04-01    46.714286
2012-04-08    45.571429
2012-04-15    30.714286
```

Min temp for every 3 days - by 3 days - 10.03/13.03/16.03

```
weather_data = weather_data.set_index("date")
```

```
weather_data.resample(pd.Timedelta(3, "day")).min_temp.mean()
```

```
date
2012-03-10    35.666667
2012-03-13    50.666667
2012-03-16    52.333333
2012-03-19    58.000000
2012-03-22    52.666667
...
2013-02-24    26.333333
2013-02-27    31.666667
2013-03-02    21.666667
2013-03-05    28.666667
2013-03-08    35.333333
Freq: 3D, Name: min_temp, Length: 122, dtype: float64
```

Or rolling - there is overlap - every day 10.03/11.03/12.03

```
weather_data.rolling(pd.Timedelta(3, "day"))
```

Rolling [window=3 days 00:00:00,min_periods=1,center=False,axis=0,method=single]

```
weather_data.rolling(pd.Timedelta(3, "day")).min_temp.mean()
```

```
date
2012-03-10    24.000000
2012-03-11    27.000000
2012-03-12    35.666667
2012-03-13    44.333333
2012-03-14    49.000000
```

Str transformations

```
coffee_data.Owner.str.upper()
```

```
0          METAD PLC
1          METAD PLC
2  GROUNDS FOR HEALTH ADMIN
3    YIDNEKACHEW DABESSA
4          METAD PLC
```

We could replace the missing data with a dummy value

```
tb.fillna(-999) # sentinel value
```

	iso2	year	m04	m514	m014	m1524	m2534	m3544	m4554	m5564	..
0	AD	1989	-999.0	-999.0	-999.0	-999.0	-999.0	-999.0	-999.0	-999.0	..
1	AD	1990	-999.0	-999.0	-999.0	-999.0	-999.0	-999.0	-999.0	-999.0	..
2	AD	1991	-999.0	-999.0	-999.0	-999.0	-999.0	-999.0	-999.0	-999.0	..

Median Conclusion:

- Use the mean when your data is normally distributed without outliers.
- Use the median when your data is skewed or contains outliers.

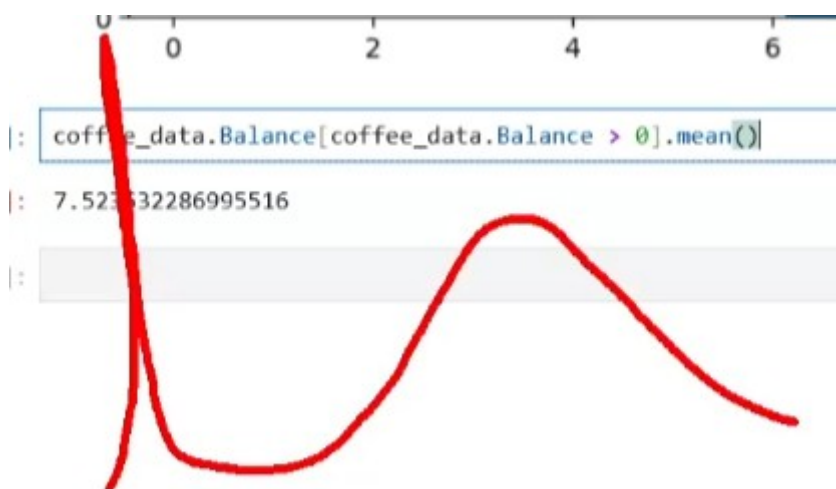
```
# imputation
coffee_data.Acidity.fillna(coffee_data.Acidity.median())
```

```
0      8.75
1      8.50
```

Mean of only positive example - when we have outliers DATASET

TRANSFORMATIONS

<https://scikit-learn.org/stable/modules/preprocessing.html#non-linear-transformation>



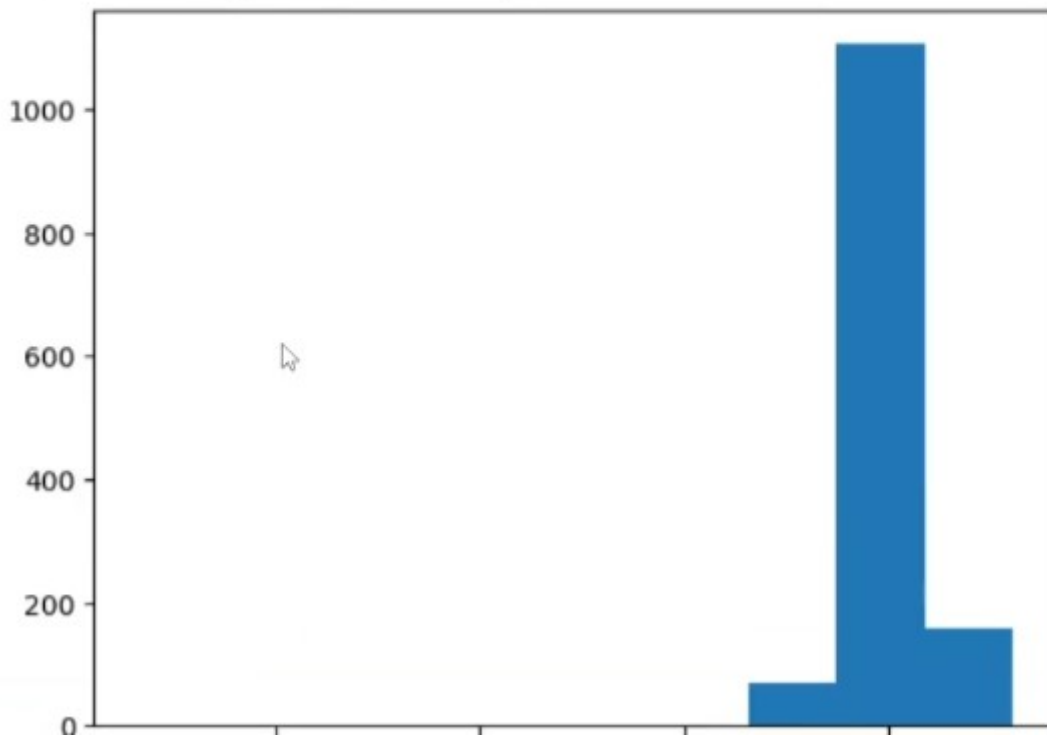
QCUT

```
pd.qcut(coffee_data.Balance, 5)
```

```
0      (7.75, 8.75]
1      (7.75, 8.75]
2      (7.75, 8.75]
3      (7.75, 8.75]
4      (7.75, 8.75]
```

```
plt.hist((coffee_data.Balance - coffee_data.Balance.mean()) / coffee_data.Balance.std())
```

```
(array([1.000e+00, 0.000e+00, 0.000e+00, 0.000e+00, 0.000e+00, 0.000e+00,  
       2.000e+00, 7.000e+01, 1.107e+03, 1.590e+02]),  
 array([-18.38399993, -16.24433896, -14.10467799, -11.96501702,  
       -9.82535605, -7.68569508, -5.54603411, -3.40637314,  
       -1.26671217,  0.8729488 ,  3.01260977]),  
 <BarContainer object of 10 artists>)
```



Find smallest 5

```
coffee_data.Balance.nsmallest(5)
```

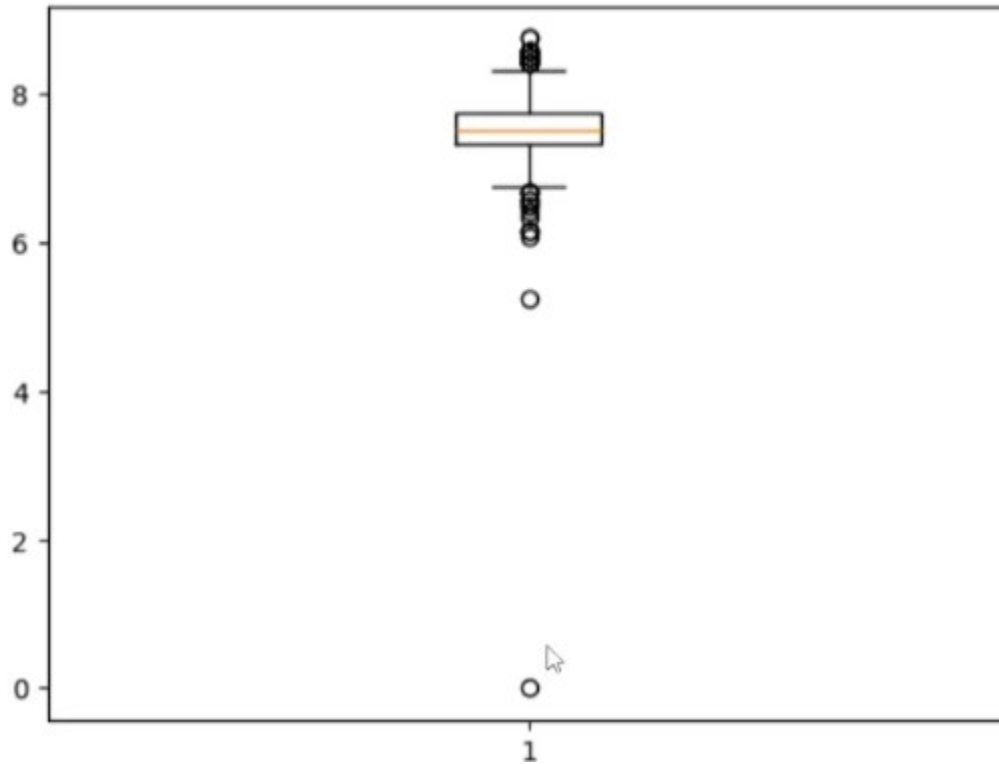
```
1310    0.00  
1335    5.25  
1308    6.08  
1303    6.17  
1304    6.17
```

Name: Balance, dtype: float64

Find outliers

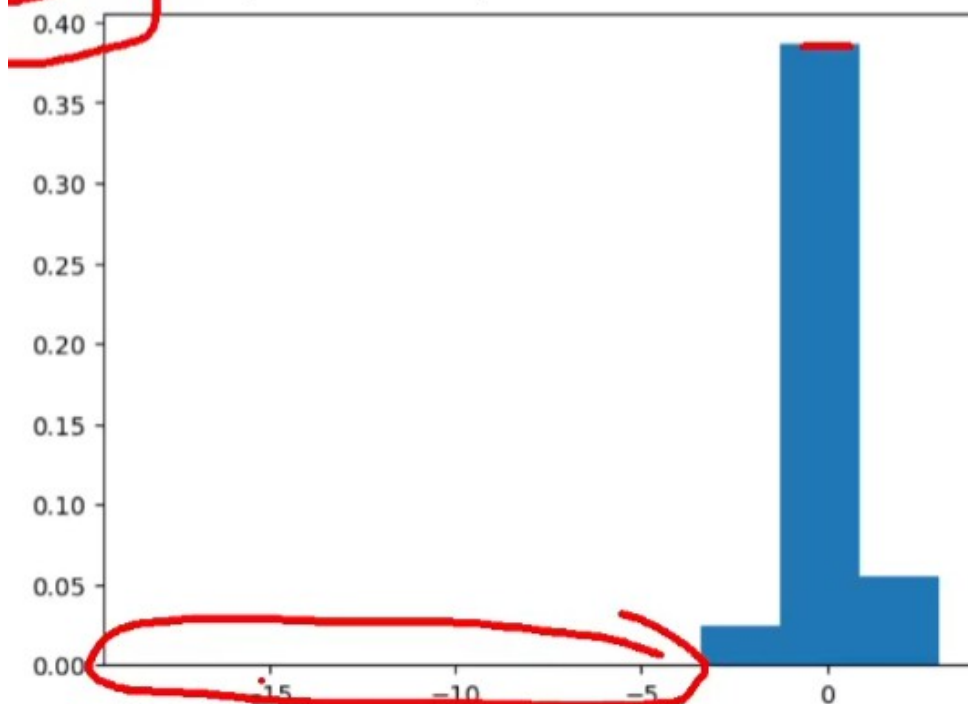
```
plt.boxplot(coffee_data.Balance)
```

```
{'whiskers': [<matplotlib.lines.Line2D at 0x1f716c8a090>,  
             <matplotlib.lines.Line2D at 0x1f71a6df210>],  
 'caps': [<matplotlib.lines.Line2D at 0x1f71ba17a90>,  
          <matplotlib.lines.Line2D at 0x1f71bae6fd0>],  
 'boxes': [<matplotlib.lines.Line2D at 0x1f718b20590>],  
 'medians': [<matplotlib.lines.Line2D at 0x1f71a6fae50>],  
 'fliers': [<matplotlib.lines.Line2D at 0x1f7194043d0>],  
 'means': []}
```




```
plt.hist((coffee_data.Balance - coffee_data.Balance.mean()) / coffee_data.Balance.std(), density = True
```

```
(array([3.49039404e-04, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
        0.00000000e+00, 0.00000000e+00, 6.98078808e-04, 2.44327583e-02,
        3.86386620e-01, 5.54972652e-02]),
array([-18.38399993, -16.24433896, -14.10467799, -11.96501702,
        -9.82535605, -7.68569508, -5.54603411, -3.40637314,
        -1.26671217, 0.8729488, 3.01260977]),
<BarContainer object of 10 artists>)
```



Median Conclusion:

- Use the mean when your data is normally distributed without outliers.
- Use the median when your data is skewed or contains outliers.

Mean of only positive example - when we have outliers DATASET

TRANSFORMATIONS

<https://scikit-learn.org/stable/modules/preprocessing.html#non-linear-transformation>

Pandas api request with authentication ChatGPT

To make an API request with authentication and load the data into a pandas DataFrame, you can use Python's requests library. Below is a step-by-step guide for handling API requests that require authentication, such as using an API key or basic authentication.

1. Install Required Libraries

If you haven't installed pandas or requests, you can install them using pip: bash

```
pip install pandas requests
```

2. Import Libraries

Start by importing the necessary libraries:

```
python Копиране на  
код  
import pandas as pd  
import requests
```

3. Set Up the Authentication

Depending on the type of authentication the API requires, you can set it up as follows: API Key in Headers

If the API uses an API key in the headers: python

Копиране на код

```
url = "https://api.example.com/data" headers = {  
    "Authorization": "Bearer YOUR_API_KEY", "Content-Type":  
    "application/json"  
}
```

```
response = requests.get(url, headers=headers) Basic
```

Authentication

If the API uses basic authentication (username and password): python

Копиране на код

```
url = "https://api.example.com/data"  
auth = ("your_username", "your_password")
```

```
response = requests.get(url, auth=auth)
```

4. Convert the JSON Response to a DataFrame

Assuming the API returns data in JSON format, convert the response to a pandas DataFrame:

```
python Копиране на  
код  
data = response.json() df =  
pd.DataFrame(data)
```

Display the DataFrame

```
print(df.head()) Example
```

Workflow

Here's an example that combines all the steps: python

Копиране на код
import pandas as pd
import requests

Example API URL

```
url = "https://api.example.com/data"
```

Headers for API Key Authentication

```
headers = {  
    "Authorization": "Bearer YOUR_API_KEY", "Content-Type":  
    "application/json"  
}
```

Make the GET request with headers

```
response = requests.get(url, headers=headers)
```

Check if the request was successful

```
if response.status_code == 200:  
    # Convert the JSON response to a DataFrame data =  
    response.json()  
    df = pd.DataFrame(data)
```

```
# Display the DataFrame  
print(df.head())
```

else:

```
print(f'Failed to retrieve data: {response.status_code}') Notes:
```

- Handling Errors: It's good practice to check the response status code before attempting to parse the JSON.
- Pagination: If the API paginates the results, you may need to loop through pages and combine the data.
- Rate Limits: Be mindful of API rate limits, and handle them appropriately (e.g., by adding delays or checking headers for rate limit information).

This method enables you to securely access API data using authentication and load it into a pandas DataFrame for further analysis.

Material Summary: Data Acquisition

1. The Scientific Method

1.1 The Scientific Method Steps

- Ask a question
- Do some research
- Form a hypothesis
- Test the hypothesis with an experiment
- Experiment works Analyze the data
- Experiment doesn't work Fix experiment
- Results align with hypothesis OK
- Results don't align with hypothesis new question, new hypothesis
- Communicate the results

2. Getting Data

2.1 The Pandas Library

- Provides a way to read and work with data
- Table (DataFrame)
- May have many dimensions
- We usually call this a "dataset"
- List (Series)
- One-dimensional
- Usually represents a column of a table
- Usage
- General requirements
- Rows and columns are indexed, columns may have names
- Each column has a fixed data type
- Python will try to infer the best type according to the data

2.2 Data Sources

- In order to work with the data, we need to represent it in tabular form
- Sometimes our data is tabular - we just need to read it
- In other cases, we need to create our tables
- Unstructured data: data that doesn't have a model
- There is some structure, it's just not very clear
- Examples: Images, plain text, audio, web pages
- Most common sources
- Tables in a text format such as .csv
- Spreadsheets (such as Excel or Google Sheets)
- Web services
- Databases

2.3 Reading a Local File

- Let's read the file accidents.csv
- Copy the file to a data folder
- Not required, just makes working with many data files easier
- Inspect the file (use a text editor or Excel) just to see what it contains
- read_csv() docs
- You'll see that all read_*() functions have a lot of optional arguments
- They make working with different formats easy, e.g.
- Instead of True and False, the table contains "Yes" and "No"
- The actual table starts at line 30 of the file
- There are blank / comment lines which should be skipped
- There are no column names in the file

2.4 Exploring the Dataset

- In Python, we can print the variable
- Even better, in Jupyter, a cell outputs its last returned value
- This will create a nicer output
- We can see that
- Rows have numerical indices starting at 0 by default
- Columns have names taken from the first line in the .csv file
- Column names:

▫ Index values:

▫ Dimensions:

▫ Format: (rows, columns)

2.5 Reading Data from Other Files

- The process is very similar
- Other text-based formats
- `pd.read_table()` is the most general function
- All others (`read_csv()`, `read_fwf()`, etc.) just apply some settings
- If we come across a file, we can apply our own settings
- The point is to match the format in the best possible way
- Example: Auto MPG dataset
- Excel
- Read the `green_tripdata_2015-09.xls` file using `pd.read_excel()`
- Explore the file dimensions

2.6 Reading Data from Web Services

- Web services work over the HTTP protocol and provide data in several formats
- Most commonly used: JSON and XML
- Some APIs to try
- Example: OpenLibrary API
- We want information about books with ISBNs
- Example: these 4 books
- We can put the URL directly, pandas will perform a GET request
- Function: `pd.read_json()`
- We can provide the parameter `orient = "index"` to arrange the dataset better
- Books should be placed by rows, their properties - by columns
- More details on this - next time
- More complex queries require more pre-processing

2.7 Reading Data from SQL

- Relational databases store data in tables
- Very similar to the datasets we use
- First, install a library to connect to databases
- From the command line:
- Then, import the library and connect to the database
- Note: This is going to vary depending on your server settings

- Perform a query

2.8 Web Scraping

- Another method for getting data
- Sometimes combined with crawling
- Traversing a Web page structure recursively
- Basic procedure
- Read a Web page as HTML
- Use the HTML to obtain the data
- A webpage is unstructured
- We need to create and maintain the structure
- We usually need more libraries to do that
- Examples
- Get all job listings from a website
- Get user contact details from a Web page

3. Using Multiple Sources

3.1 Data Guidelines

- Some queries will not be simple
- E.g., scraping, dealing with "freeform" text, audio data, networks
- We need to create a tabular structure from the raw data

- How? We'll discuss this later in the course
- After we read the data, we have to ensure it's been read without errors
- A very simple first check: check the dimensions (`dataframe.shape`) and show the first few rows (`dataframe.head()`)
- We may need to rename columns
- We may need to perform different manipulations to ensure the data is in a proper state
- We'll do this in the next lectures

3.2 Merging Many Data Sources

- Automate the process as much as possible
- From reading the raw data to getting the processed dataset
- If the dataset changes or updates, you'll just re-run your code
- Document the process
- Create as few datasets as possible
- I.e., merge many sources into one table if you can
- We'll talk more about combining relations next time
- Ensure the different sources are compatible and consistent
- If they aren't, process the raw data
- Most common example: Mismatched IDs
- Make sure all column types are correct
- Check: `dataframe.dtypes`
- Example: `str` type for a numeric column

Material Summary: Data Tidying and Cleaning

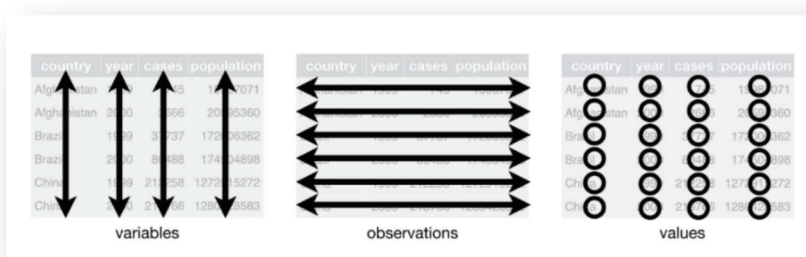
1. Data Tidying

1.1 Tidy Data

- Most important rules when creating (or using) datasets
 - Columns – attributes (features, variables)
 - Rows – observations
 - Cells – values (one observation of one feature)
 - All other data is called **messy data**
- Empirical rule for testing whether a dataset is tidy
 - Adding one more observation should create one new row
 - No new columns
 - No multiple rows
 - No partial rows, no changes to other rows
- pandas allows us to read, tidy up and transform datasets
 - Data modelling requires a tidy and clean dataset in order to work well (garbage in – garbage out)

1.2 Messy Data

- What we want



- What we get instead

1.3 Tidy and Messy Data

- A very good [paper](#) on tidy data
- Example: several datasets

country year cases population				Tidy dataset			
1	Afghanistan	1999	745	19987071	1	Afghanistan	1999
2	Afghanistan	2000	2666	20595360	2	Afghanistan	1999
3	Brazil	1999	37737	172006362	3	Afghanistan	2000
4	Brazil	2000	80488	174504898	4	Afghanistan	2000
5	China	1999	212258	1272915272	5	Brazil	1999
6	China	2000	213766	1280428583	6	Brazil	1999

country year rate							
1	Afghanistan	1999	745/19987071	7	Brazil	2000	cases
2	Afghanistan	2000	2666/20595360	8	Brazil	2000	population
3	Brazil	1999	37737/172006362	9	China	1999	cases
4	Brazil	2000	80488/174504898	10	China	1999	population
5	China	1999	212258/1272915272	11	China	2000	cases
6	China	2000	213766/1280428583	12	China	2000	population

- Same information, different ease of use

1.4 Messy to Tidy Data

- The table header contains values
 - Identify the variables and distribute (unpivot) the values
- Read the pew.csv dataset
 - Distribution of income by religion
- Show the first 5 values (use the head() function)
 - Also see the number of variables and observations (shape)
 - This will also ensure that you've read the dataset correctly
 - Variables:** religion, income, frequency
- Transform the dataset to make it tidy ([docs](#))

```
pew = pd.read_csv("pew.csv")
pew_tidy = pew.melt(
    id_vars = ["religion"], # Identifier variables (all others are "unpivoted")
    var_name = "income", # Variable
    value_name = "frequency" # Value
)
```

- Multiple variables stored in one column
 - Identify and split the variables into separate columns
- Read the tb.csv dataset
 - Tuberculosis cases
 - m04, m514, m1524, etc. contain two variables (gender and age)
 - male, 0-4 years old; male, 5-14 years old, etc.
 - There's also a problem with missing values (NaN)
- Tidying process
 - First, melt all columns (they are values and should not be)
 - Next, split the column names and extract the gender and age information
 - Add the new info to the dataset

- Remove all missing values

```
def process_age_group(age_group):
    ages = {"04": "0-4", "65": "65+", "u": "unknown"}
    if age_group in ages:
        return ages[age_group]
    else:
        # Put a dash before the last two digits
        return f"{age_group[:-2]}-{age_group[-2:]}"

tb = tb.melt(
    id_vars = ["iso2", "year"], var_name = "sex_and_age", value_name = "cases")

tb["sex"] = tb.sex_and_age.str.get(0)
tb["age_group"] = tb.sex_and_age.str.slice(1)
tb = tb.drop(columns = "sex_and_age")

tb.age_group = tb.age_group.apply(process_age_group)

# Tidy up the column and row order
tb = tb[["iso2", "year", "sex", "age_group", "cases"]]
tb = tb.sort_values(["iso2", "year"])
```

3. Variables are stored in both rows and columns

- Identify and split the variables
- Read the weather.csv dataset
 - Daily weather records in Mexico in 2010
 - d1, d2, etc. are the days of a month; tmin and tmax should be columns
 - Make a new column with the date: [date, tmin, tmax]
- Tidying process
 - Melt all days
 - Create days based on date, month and year
 - Pivot the tmin and tmax columns

```
weather_data = weather_data.melt(
    id_vars = ["id", "year", "month", "element"], var_name = "day")
weather_data.day = weather_data.day.str.slice(1).astype(int)

# Remove missing / invalid days (e.g., 31st April) and dates with no records
weather_data = weather_data.dropna()
weather_data["date"] = pd.to_datetime(weather_data[["year", "month", "day"]])
weather_data = weather_data.drop(columns = ["year", "month", "day"])

# Pivot the elements back to their own columns
weather_data = weather_data.pivot_table(
    index = ["id", "date"], columns = "element", values = "value")

# Pivoting returns a multi-indexed element, go back to a flat DataFrame
weather_data = weather_data.reset_index()
weather_data.columns.name = ""
weather_data = weather_data[["id", "date", "tmin", "tmax"]]
```

4. One type in multiple tables

- Merge the tables into one
 - Read all tables, add the new columns
 - Often the filename should be in its own column (if it's important)
 - Melt and tidy if necessary

5. Multiple types in one table

- Split into more tables
 - If necessary, introduce relations (similar to a relational database)
- Each table should be responsible for one type of measurement
- * Read the billboard.csv dataset and apply those transformations

2. Operations on Datasets

2.1 Subsetting Rows

- Selecting only some rows (aka **selection**)
- First / last n records (observations)

```
weather_data.head(10)
weather_data.tail() # 5 by default
```

- Random n records

```
weather_data.sample(n = 10)
weather_data.sample() # 1 random record by default
```

- Smallest / largest n records in each column
- Subsetting by a Boolean expression (predicate)
 - Returns only rows where the expression returns True

```
weather_data[weather_data.tmax > 30]
weather_data.nsmallest(3, "tmax")
weather_data.nlargest(3, "tmax")
```

- Selecting only some columns (aka **projection**)
- Single column (returns a Series object)

```
weather_data["tmax"]
weather_data.tmax # Possible in most cases
```

- More than one column (returns a DataFrame object)

```
weather_data[["tmin", "tmax"]]
```

- Combining filters

```
weather_data[weather_data.date > "2010-08-01"][["date", "tmax"]]
weather_data.loc[weather_data.date > "2010-08-01", ["date", "tmax"]]
```

- A note on Boolean expressions
 - and, or, not are &, |, ~
 - **Always** put parentheses around the individual expressions

```
weather_data[
    (weather_data.date > "2010-08-01") & (weather_data.date < "2010-09-01")]
```

- These methods work by columns
 - If multiple columns are passed, they are applied to each column individually

```
print("Count:", weather_data.tmin.count()) # number of non-null values
print("Min:", weather_data.tmin.min())
print("Max:", weather_data.tmin.max())
print("Mean:", weather_data.tmin.mean())
print("Median:", weather_data.tmin.median())
print("Standard deviation:", weather_data.tmin.std())
```

- Grouping
 - Splits the data into several groups based on the values of a column
 - We have to apply a method after grouping
 - Or iterate over the groups (using a for-loop)
 - Example: Average number of people for each income group

```
pew_tidy.groupby("income").mean()
```

3. Cleaning Data

3.1 Cleaning Data

- No common way of doing this
- We need to rely on intuition and some common patterns
 - Tidy up the dataset
 - You must know the dataset documentation first
 - Treat nulls / NaNs: either remove them or replace them
 - Replacing values might be **dangerous**
 - If done properly, it will affect the data in a positive way
 - Identify and fix errors (also **dangerous**)
 - Melt and pivot datasets
 - Merge (join) and separate datasets
 - Subset variables and / or observations
 - Summarize and group variables
 - [Pandas Cheat Sheet](#)

3.2 Example: Weather Data

- Since there's no common way of cleaning, we'll explore and clean a dataset, showing steps and examples as we go
- [Dataset](#) (weather data, courtesy of synesthesiam@github)
- Read the dataset (you don't need to download it)
 - See how many variables and observations are there
 - Display the first and last few rows to get a sense of the data
 - Check the data types (to see if something's wrong with the reading)
 - E.g., numbers recognized as strings
 - See a subset of the columns
 - Summarize (describe) the dataset
- The column names don't look good
 - Make them "pythonic" (lowercase_with_underscores)
 - This will make selecting them easier (weather.mean_temp)

```
weather.columns = ["date", "max_temp", "mean_temp", "min_temp", "max_dew",  
                  "mean_dew", "min_dew", "max_humidity", "mean_humidity",  
                  "min_humidity", "max_pressure", "mean_pressure",  
                  "min_pressure", "max_visibility", "mean_visibility",  
                  "min_visibility", "max_wind", "mean_wind", "max_gusts",  
                  "precipitation", "cloud_cover", "events", "wind_dir"]
```

- What are the ranges of data?
 - E. g. temperature, pressure, humidity
 - Use the min() and max() methods
- * Try to explore the data a bit
 - Plot a few histograms and / or boxplots to see the distributions
- Convert the dates to a datetime object
 - To make performing time-dependent analysis easier

```
weather.date = pd.to_datetime(weather.date)
```

- If needed, use apply() to perform a function on every row

```
from datetime import datetime  
def string_to_date(date_string):  
    return datetime.strptime(date_string, "%Y-%m-%d")  
  
weather.date = weather.date.apply(string_to_date)
```

- It's even better to use dates as indices (when we need to subset date ranges or perform other time-dependent tasks)

```
weather = weather.set_index("date") # or use inplace = True  
  
print(weather.loc[pd.to_datetime("2012-08-19")])  
# or weather.loc["2012-08-19"], or any other formatting
```

- Also see why precipitation is not a float and edit it
- Remove or replace missing values
 - In this case, replacing is better because removing takes away an entire row

```
weather_with_events = weather.dropna(subset = ["events"])  
weather.events = weather.events.fillna("") # Better
```

- Try to see how variables interact – group the data
 - E.g., by cloud cover and events
 - Print the number of days when each combination of {cover, events} occurred

```
for (cover, events), group_data in weather.groupby(["cloud_cover", "events"]):  
    print(f"Cover: {cover}, Events: {events}, Count: {len(group_data)}")  
# Or: weather.groupby(["cloud_cover", "events"]).size()
```

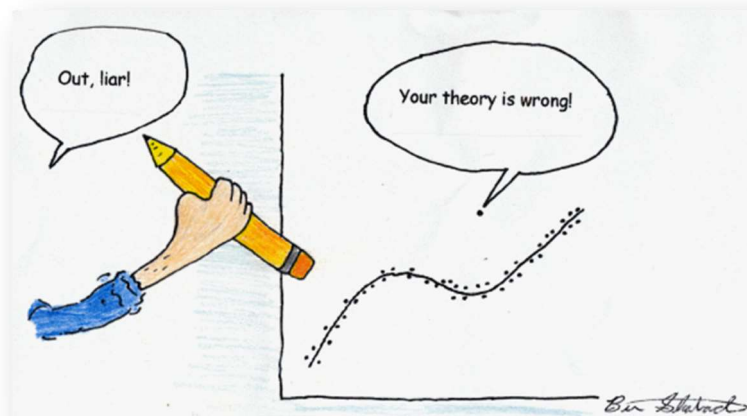
- Plot data
 - Next time

- If needed, perform transformations
 - Math operations: log, square root, addition, multiplication, etc.
 - Be careful as you'll get results in different dimensions
 - Normalizing scores (such as using Z-scores) is recommended in most cases
 - It's much better for ML algorithms to have data of similar scales
 - You can do that manually or use a library (such as [sklearn.preprocessing](#))
 - By convention, calculated columns are added to the dataset
- **Describe all operations as you're doing them**
 - Describe what you're doing and why
 - Useful to check your work later (or allow others to do that)
 - If needed, save the resulting dataset into a file
 - Supply your data transformation log with it
 - Provide a dataset description

3.3 Outliers and Errors

- **Outliers** – values which are far from their expected range
 - Or having a very low probability of happening (assuming a model)
- Many possible cases
 - Wrong data entry (e.g. an adult weighing 5kg might be 50kg or something else)
 - Wrong assumptions (the data is correct, our view isn't)
- What to do?
 - Inspect the data point
 - Try to figure out what happened
 - If needed, remove the row or try to replace the value
 - Try a transformation
 - If possible, perform analysis with and without the outlier(s) and compare your results

3.4 Transformations on Features



- The quality of our results depends strongly on the features we use
 - "Garbage in – garbage out"
- **Dimensionality reduction**
 - Reducing the number of variables (features)
 - We can do this manually or use algorithms
 - **Feature selection**
 - Selecting only columns that are useful
 - **Feature extraction**
 - Transforming non-structured to structured data
 - Examples: images, audio, text
 - Getting meaningful features
- **Feature engineering**
 - Using our knowledge of the data to create meaningful features
 - Involves a lot of brainstorming and testing

3.5 Next Steps (Optional)

- Have a look at scikit-learn's ["Dataset Transformations"](#) module
 - It describes the most common operations
 - Data cleaning
 - Dimensionality reduction

- Feature extraction
- There are many algorithms based on
 - Data types (e.g., text or numerical data, labelled vs. not labelled)
 - Model types (how we want to present our data, e.g., linear model)
 - Algorithm types (e.g., finding similar news articles, recommending movies to users, classifying, etc.)
- No "hard and fast rule", use your intuition
 - Knowing more tools / models / algorithms -> better performance