# 2. Extract, Transform, Load

## Extract

For this lesson, extract the zipped folder "data_02.zip" from "Week 1: Day 2", and place the folder in your project directory.

## Different File Locations

### Local files

```
import pandas as pd

df = pd.read_csv("data_02/country_data_index.csv")
```

### Files in a different location

```
import pandas as pd
```

```
df = pd.read_csv("data_02/country_data_index.csv")
```

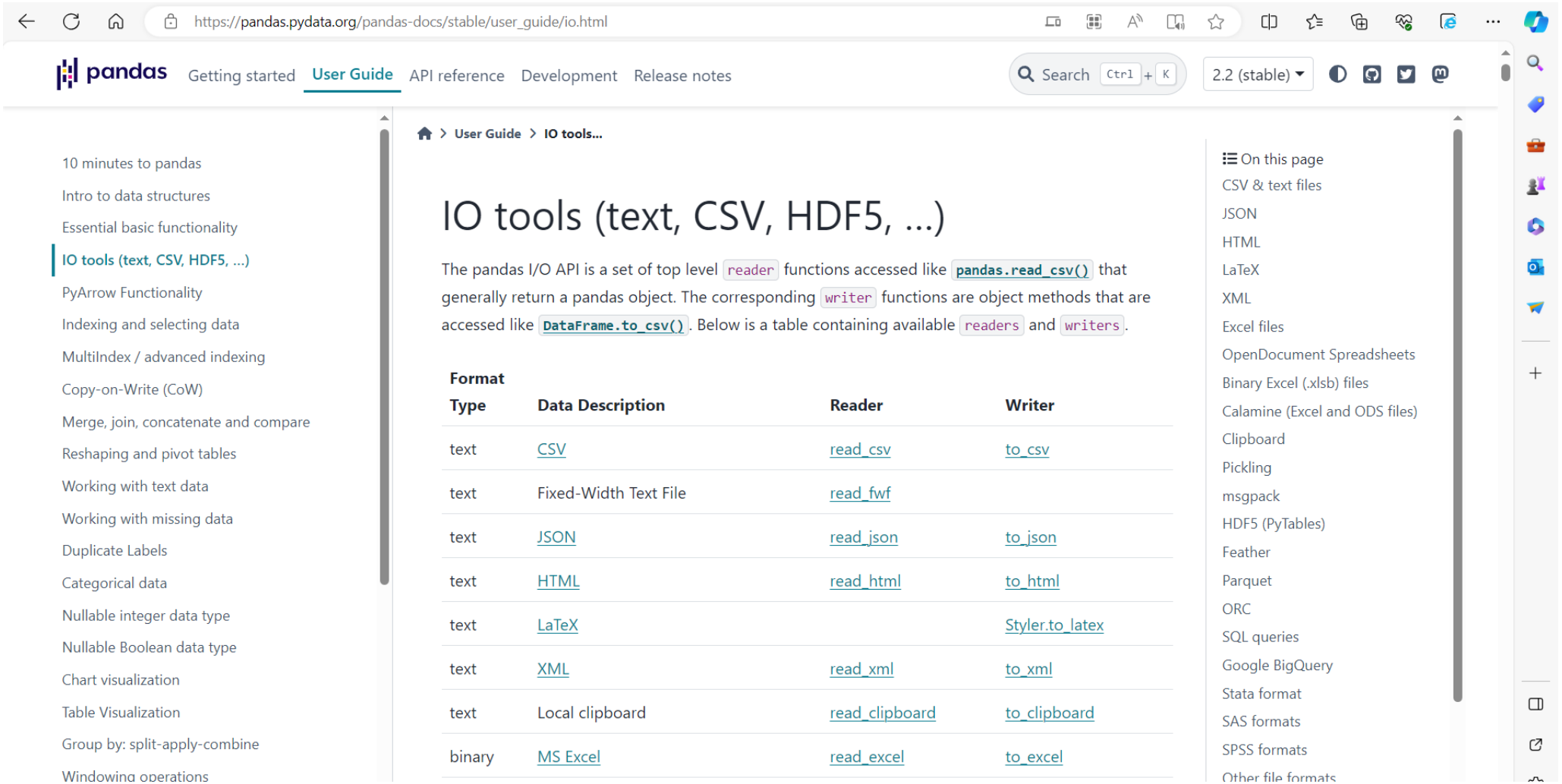
## File from a URL

```
import pandas as pd

df = pd.read_csv("https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.dat
a")
```

This data has no header information for each columns, i.e no column names. Let us add some:

```
column_names = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width', 'class']

df = pd.read_csv("https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.dat
a",header=None, names= column_names)
```

# Different File Types




https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html

# Text Files

Text file with semi-colon

```
df = pd.read_csv("data_02/Geospatial Data.txt",sep=";")
```

### Excel

```
df = pd.read_excel("data_02/residentdoctors.xlsx")
```

### Json

```
df = pd.read_json("data_02/student_data.json")
```

## Other options:

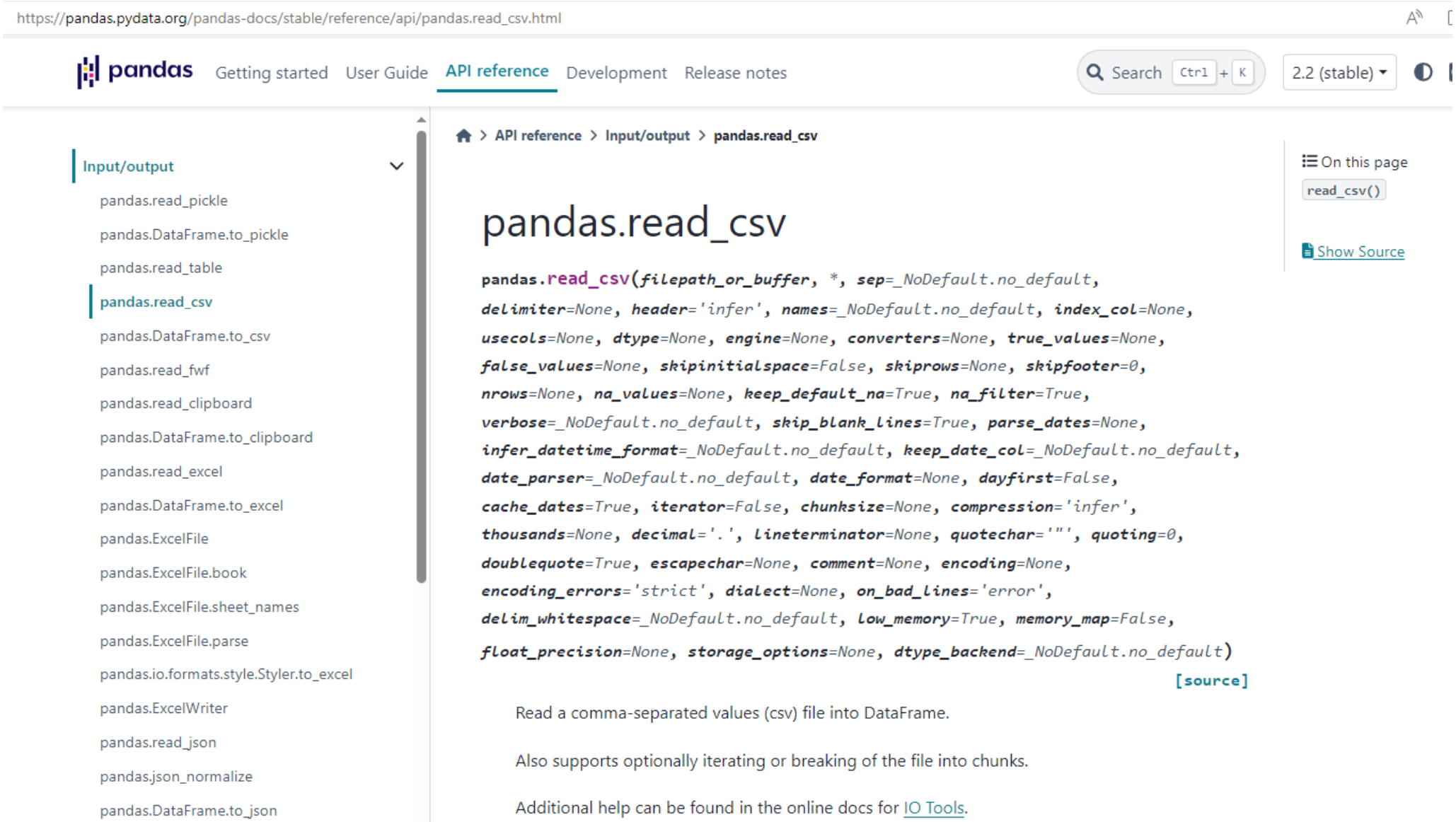
> *Would you like to see the following:*

- Webscraping
- Connecting to Databases

> *Upload some of your own data files to the #chat channel in slack, we can analyze them too!*

## Transform

As we saw in the previous lesson, sometimes the data is not in the correct format. There are many ways to clean or transform files. Sometimes this is done in the "Extract" process too, so there is an overlap. Here we will recap on some parameters that the the read*csv function has: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read*csv.html




Let us look at a few files with some small issues again.

# Index Column

```
df = pd.read_csv("data_02/country_data_index.csv")
```

The "Unnamed: 0" column often appears in pandas DataFrames when reading a CSV file if the CSV file itself has an index column that was not explicitly specified.

When you read a CSV file into a DataFrame using pd.read_csv(), pandas assigns an index to each row by default. If the CSV file already contains an index column, pandas will create a new index unless you specify the existing column as the index. If no index is specified, pandas will create a default numerical index, and this default index is labeled as "Unnamed: 0" in the DataFrame.

To avoid the appearance of the "Unnamed: 0" column, you can either specify the existing index column from your CSV file when reading the data, or you can use the index_col parameter to explicitly specify which column you want to use as the index.

```
df = pd.read_csv("data_02/country_data_index.csv",index_col=0)
```

This will read the CSV file and use the first column as the index, preventing the appearance of the "Unnamed: 0" column in your DataFrame.

# Skip Rows

As we saw in the first lesson:

```
df = pd.read_csv("data_01/insurance_data.csv")
```

We get an error because the number of columns in your CSV file is not consistent across all rows. The read_csv function in pandas assumes a consistent number of columns, but your data seems to have two values in some rows, causing the error.

We can resolve this by using the "skiprows" parameter to just skip first 5 rows:

```
df = pd.read_csv("data_02/insurance_data.csv",skiprows=5)
```

# Column Headings

```
df = pd.read_csv("data_02/patient_data.csv")
```

No headings :(

Create a list with the heading names you want and add use the parameters "header=None" and "names=column_names":

```
column_names = ["duration", "pulse", "max_pulse", "calories"]

df = pd.read_csv("data_02/patient_data.csv", header=None, names=column_names)
```

# Unique Delimiter

As we saw in the first lesson:

```
df = pd.read_csv("data_02/Geospatial Data.txt")
```

This is a text file with the data separate with a semi-colon, Pandas is expecting a comma. So we use the parameter sep=";"

```
df = pd.read_csv("data_02/Geospatial Data.txt",sep=";")
```

Also note that the file name has a space in it, which is not recommended.

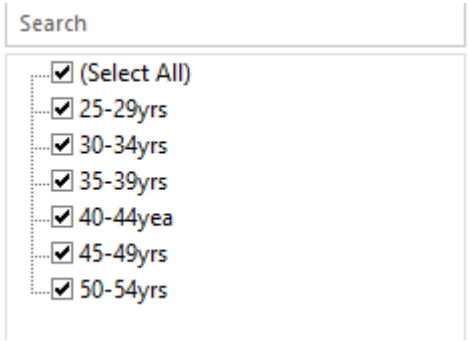
# Inconsistent Data Types & Names

```
df = pd.read_excel("data_02/residentdoctors.xlsx")
```

- Some column headings are snake case and other all caps
- AGEDIST has numbers with text
- MARITULSTATUS has text fields

Note with Excel files, you cannot have it open in Spyder and in Excel too.

Let us first look at the different groupings:




One way to make this more efficient is to convert it the lower age in grouping. So instead of 30-34yrs it would be 34. This is just one way. You can keep it as is however it will prevent you from doing further numerical analysis on it.

Add the following code:

```
# Step 1: Extract the lower end of the age range (digits only)
df['LOWER_AGE'] = df['AGEDIST'].str.extract('(\d+)-')
```

Pseudo-code: 1. Search for a number followed by a hyphen like "30-" 2. If you find that number, extract the number and ignore the hyphen 3. Put it in a new column called LOWER_AGE

Then convert that number to from s tring to integer, a whole number.

This step uses "str.extract()" to capture one or more digits from the 'AGEDIST' column. When you use `df['AGEDIST'].str.extract('(\d+)-')`, it applies this regular expression pattern to each element in the 'AGEDIST' column.

Regular expressions (regex or regexp) are sequences of characters that define a search pattern. It extracts the portion of the string that matches the pattern, which is the one or more consecutive digits before the hyphen in the age range. We won't go over regular expressions much, this was just a demonstration to show you different possibilitles.

```
# Step 2: Convert the new column to float
df['LOWER_AGE'] = df['LOWER_AGE'].astype(int)
```

Pseudo-code: 1. Convert all the data to integers 2. Store that value back into LOWER_AGE column

After extracting the digits, this step converts the resulting column to floating-point numbers using astype(float). Without this conversion, the extracted values would be treated as strings, and any subsequent numeric operations or analysis would not work as expected.

### Pandas Functions/Methods

In pandas, .str, .extract(), and .astype() are all functions or methods that can be applied to a pandas Series object or single column of text data.

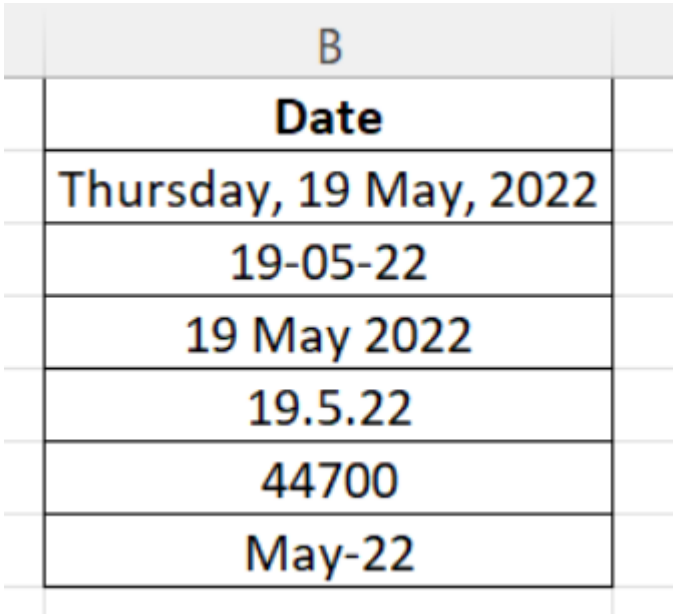
These methods are part of the powerful functionality provided by pandas for manipulating and analyzing data in a DataFrame.

When you apply .str to a column, it allows you to perform various string operations on each element of the Series. Here we used .extract() to extract certain information from a string. Other string methods: - .upper() makes string/text upper case - .lower() makes string/text lower case - .replace() replaces certain characters with another

*Note, they all have brackets as they can accept parameters.*

## Working with Dates

Again there are many ways to do things, sometimes the recommendations are useful and sometimes they are not. Working with dates and time is a tricky subject, as it all depends what you want to do with it. One general problem with dates in Excel is that formatting can be completely different from one laptop to another, you may send an Excel spreadsheet to someone else and the dates maybe completely different which may effect the analysis you did. For example:



| B |
| --- |
| **Date** |
| Thursday, 19 May, 2022 |
| 19-05-22 |
| 19 May 2022 |
| 19.5.22 |
| 44700 |
| May-22 |

Ideally you would want to work with a date format that is completely clear for yourself and the next person working with it. Let us look at this time series data:

```
df = pd.read_csv("data_02/time_series_data.csv")
```

You will notice the date is in this format: 2020-01-10, however in the US, the day is the middle number and in UK and in most local countries here the middle number is the month - as it should be :)

Another consideration is that dates are usually interpreted as strings or text in a csv file. This can be checked with the .info() method. So the first thing you need to do is convert it to date format using Pandas.

```
# Convert the 'Date' column to datetime
df['Date'] = pd.to_datetime(df['Date'])
```

For example, if your date string is in the "DD-MM-YYYY" format, you would specify the format like this:

```
df['Date'] = pd.to_datetime(df['Date'], format='%d-%m-%Y')
```

%d: day %m: month %Y: year

This not only makes the date column unambiguous but also enables various date-related operations and manipulations. For example:

- filtering by date or range
- calculating time difference
- extracting year, month, day
- add shifts or lags to data
- rolling windows
- timezones
- date arithmetic

Another option is to split the Date column into separate columns, i.e. year, month and day. For example:

```
# Split the 'Date' column into separate columns for year, month, and day
df['Year'] = df['Date'].dt.year
df['Month'] = df['Date'].dt.month
df['Day'] = df['Date'].dt.day
```

Output:

```
        Date  Temperature  Year  Month  Day
0 2020-01-01        27.48  2020      1    1
1 2020-01-02        24.31  2020      1    2
2 2020-01-03        28.24  2020      1    3
3 2020-01-04        32.62  2020      1    4
4 2020-01-05        23.83  2020      1    5
5 2020-01-06        23.83  2020      1    6
6 2020-01-07        32.90  2020      1    7
7 2020-01-08        28.84  2020      1    8
...
```

There may be cases when you need to do this for example:

- time series analysis
- seasonal analysis
- grouping and aggregation
- feature engineering for machine learning
- database filtering
- enhance data integrity

## NANs and Wrong Formats

Import the following data:

```
import pandas as pd

df = pd.read_csv('data_02/patient_data_dates.csv')

# Allows you to see all rows
pd.set_option('display.max_rows',None)

print(df)
```

Output

```
 Index Duration Date Pulse Maxpulse Calories
0 0 60 '2020/12/01' 110 130 409.1
1 1 60 '2020/12/02' 117 145 479.0
2 2 60 '2020/12/03' 103 135 340.0
3 3 45 '2020/12/04' 109 175 282.4
```

```
4 4 45 '2020/12/05' 117 148 406.0
5 5 60 '2020/12/06' 102 127 300.0
6 6 60 '2020/12/07' 110 136 374.0
7 7 450 '2020/12/08' 104 134 253.3
8 8 30 '2020/12/09' 109 133 195.1
9 9 60 '2020/12/10' 98 124 269.0
10 10 60 '2020/12/11' 103 147 329.3
11 11 60 '2020/12/12' 100 120 250.7
12 12 60 '2020/12/12' 100 120 250.7
13 13 60 '2020/12/13' 106 128 345.3
14 14 60 '2020/12/14' 104 132 379.3
15 15 60 '2020/12/15' 98 123 275.0
16 16 60 '2020/12/16' 98 120 215.2
17 17 60 '2020/12/17' 100 120 300.0
18 18 45 '2020/12/18' 90 112 NaN
19 19 60 '2020/12/19' 103 123 323.0
20 20 45 '2020/12/20' 97 125 243.0
21 21 60 '2020/12/21' 108 131 364.2
22 22 45 NaN 100 119 282.0
23 23 60 '2020/12/23' 130 101 300.0
24 24 45 '2020/12/24' 105 132 246.0
25 25 60 '2020/12/25' 102 126 334.5
26 26 60 26 12 2020 100 120 250.0
27 27 60 '2020/12/27' 92 118 241.0
28 28 60 '2020/12/28' 103 132 NaN
29 29 60 '2020/12/29' 100 132 280.0
30 30 60 '2020/12/30' 102 129 380.3
31 31 60 '2020/12/31' 92 115 243.03
```

Now you will notice the following:

1. The data set has an index column that is redundant
2. The data set contains some empty cells or NaNs (“Date” in row 22, and “Calories” in row 18 and 28, “Maxpulse” in row 1).
3. The data set contains wrong format (“Date” in row 26).
4. The data set contains wrong data (“Duration” in row 7 and 13).
5. The data set contains duplicates (row 11 and 12).

### Remove the columns

Now the index column is redundant and we do not need it. We can remove it with drop method in the following way:

```
df.drop(['Index'],inplace=True,axis=1)
```

This method is true for any column you want to remove.

### Replace Empty Values - Using fillna

Before removing rows completely because of a NaN, it is better to try to first modify to work with the data. So we can insert a new value instead. This way you do not have to delete entire rows just because of some empty cells as the other columns may have useful data. The fillna() method allows us to replace empty cells with a value. A common way to replace empty cells, is to calculate the mean (average), median (middle) or mode (most frequent) value of the column. Pandas uses the mean(), median() and mode() methods to calculate the respective values for a specified column:

```
x = df["Calories"].mean()

df["Calories"].fillna(x, inplace = True)
```

When inplace=True, it means that the changes made by the fillna operation will be applied directly to the original DataFrame (df in this case), and it will not return a new DataFrame. The inplace parameter modifies the DataFrame in place, without the need to assign the result back to a variable.

### Wrong Date Format – Convert with to_datetime()

Cells with data of wrong format can make it difficult, or even impossible, to analyze data. To fix it, you have two options: remove the rows, or convert all cells in the columns into the same format. In our Data Frame, we have two cells with the wrong format. Check out row 22 and 26, the 'Date' column should be a string that represents a date. The best way to fix the date is to convert it a versatile date time format unique to Pandas using to_datetime():

```
df['Date'] = pd.to_datetime(df['Date'])
```

Try it out check that the dates look correct. As you can see from the result, the date in row 26 was fixed, but the empty date in row 22 got a NaT (Not a Time) value, in other words an empty value. One way to deal with empty values is simply removing the entire row using df.dropna(inplace = True).

If you only want to remove rows in that Date column you use df.dropna(subset=['Date'], inplace = True).

## Removing Empty Cells – Using dropna

Empty cells can potentially give you a wrong result when you analyze data. One way to deal with empty cells is to remove rows that contain empty cells. This is usually OK, since data sets can be very big, and removing a few rows will not have a big impact on the result. This can be done using the df.dropna() function. By default, the dropna() method returns a new DataFrame, and will not change the original.

If you want to change the original DataFrame, use the inplace = True argument. The dropna(inplace = True) will NOT return a new DataFrame, but it will remove all rows containing NULL values from the original DataFrame. You will also need to reset the index with df.reset_index(drop=True) as if you remove a row, the row numbers will not be consecutive:

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

## Wrong Data – Replace and Remove Rows

“Wrong data” does not have to be “empty cells” or “wrong format”, it can just be wrong, like if someone registered “199” instead of “1.99”. Sometimes you can spot wrong data by looking at the data set, because you have an expectation of what it should be. If you take a look at our data set, you can see that in row 7, the duration is 450, but for all the other rows the duration is between 30 and 60.

It doesn’t have to be wrong, but taking in consideration that this is the data set of someone’s workout sessions, we conclude with the fact that this person did not work out in 450 minutes.

One way to fix wrong values is to replace them with something else.

In our example, it is most likely a typo, and the value should be "45" instead of "450", and we could just insert "45" in row 7 with

```
df.loc[7, 'Duration'] = 45:
```

Alternatively, you could have removed that row completely using df.drop(7, inplace = True). For small data sets you might be able to replace the wrong data one by one, but not for big data sets.

### Removing Duplicates – Using drop_duplicates()

Duplicate rows are rows that have been registered more than one time. By taking a look at our test data set, we can assume that row 11 and 12 are duplicates. To discover duplicates, we can use the duplicated() method. The duplicated() method returns a Boolean values for each row. To remove the duplicates use drop_duplicates() method:

```
df.drop_duplicates(inplace = True)
```

### Final Cleaned Code

```
import pandas as pd

df = pd.read_csv('data_02/patient_data_dates.csv')

pd.set_option('display.max_rows',None)
```

```
print(df)

# Drop Index Column:

df.drop(['Index'],inplace=True,axis=1)

print(df)

# Fill NaNs or empty fields in Calorie Column

x = df["Calories"].mean()

df["Calories"].fillna(x, inplace = True)

print(df)

# Convert Wrong Date Format in Date Column

df['Date'] = pd.to_datetime(df['Date'])

# Drop NaT field in Date Column

df.dropna(subset=['Date'], inplace = True)

print(df)

# Remove any rows that have NaNs or empty fields
# Here only the row 1 for the MaxPulse column as the rest have been resolved
```

```
df.dropna(inplace = True)

# Reset index
df = df.reset_index(drop=True)

print(df)

# Remove duplicates found in line 10 and 11
df.drop_duplicates(inplace = True)

df = df.reset_index(drop=True)

print(df)
```

# Applying Data Transformations

Now we will look at more involved data transformation methods, namely aggregations, appending, merging, and filtering. We will demonstrate how to:

- aggregate data using groupby in pandas
- append and merge datasets using different join types
- filter and manipulate data to create new variables.

## Aggregation

```
grouped = df.groupby('class')
```

```
# Calculate mean, sum, and count for the squared values
mean_squared_values = grouped['sepal_length_sq'].mean()
sum_squared_values = grouped['sepal_length_sq'].sum()
count_squared_values = grouped['sepal_length_sq'].count()

# Display the results
print("Mean of Sepal Length Squared:")
print(mean_squared_values)

print("\nSum of Sepal Length Squared:")
print(sum_squared_values)

print("\nCount of Sepal Length Squared:")
print(count_squared_values)
```

In Pandas, the groupby function is used to group data based on some criteria, and then you can perform various operations on each group.

## Append & Merge

We will first look at appending two data sets together that have the same column names. For example "person*split1.csv" and "person*split2.csv:

```
import pandas as pd

# Read the CSV files into dataframes
df1 = pd.read_csv("data_02/person_split1.csv")
df2 = pd.read_csv("data_02/person_split2.csv")
```

```
# Concatenate the dataframes
df = pd.concat([df1, df2], ignore_index=True)
```

What happens if you two tables with different column names but are related, i.e relational data. For example in the one csv file we have : "id, Company, Name, Department, Job, Title, Skill" as column names and the other: "id, University". If you are familiar with SQL databases then you should know about this.

So if we want to merge these two datasets together. They are related by the "id" column. We can do that with the following code:

```
df1 = pd.read_csv('data_02/person_education.csv')
df2 = pd.read_csv('data_02/person_work.csv')

## inner join
df_merge = pd.merge(df1,df2,on='id')
```

So in this case we used the pandas merge method to join them both based on their "id" column. Here, by default an inner join was used.

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.merge.html

# pandas.merge

```
pandas.merge(left, right, how='inner', on=None, left_on=None, right_on=None,
left_index=False, right_index=False, sort=False, suffixes=('_x', '_y'),
copy=None, indicator=False, validate=None)                          [source]
```

Merge DataFrame or named Series objects with a database-style join.

A named Series object is treated as a DataFrame with a single named column.

The join is done on columns or indexes. If joining columns on columns, the DataFrame indexes *will be ignored*. Otherwise if joining indexes on indexes or indexes on a column or columns, the index will be passed on. When performing a cross merge, no column specifications to merge on are allowed.

An inner join returns only the rows where there is a match in both dataframes on the specified "on" column (in this case, the "id" column). If there is no match, the row is excluded from the result.

For example, here an outer join is used

```
df_merge = pd.merge(df1, df2, on='id', how='outer')
```

An outer join returns all the rows from both dataframes. If there is no match for a row in either dataframe, the missing values will be filled with NaNs. Left and Right Joins are possible too.

# Filtering

We saw in the first lesson using the "country_data.csv" we can fitler simply with the following:

```
# Filtering data
print(df[df['age'] > 30])
```

In a similar way we can filter by a certain category and calculate its mean for "sepal-length". Look at the "iris.csv" data set. Let us say we only want the "Iris-versicolor" category. We can say:

```
# Filter data for females (class == 'Iris-versicolor')
iris_versicolor = df[df['class'] == 'Iris-versicolor']

# Calculate the average iris_versicolor_sep_length
avg_iris_versicolor_sep_length = iris_versicolor['sepal_length'].mean()
```

> _When we want to do check for a comparison we use double equals "==" not a single equals sign "="

There is also a better way to label the "class" column since the word "Iris-" is redundant. We can remove it in the following way:

```
df['class'] = df['class'].str.replace('Iris-', '')
```

If you have your own custom change you want to do to each value:

```
# Apply the square to sepal length using a lambda function
df['sepal_length_sq'] = df['sepal_length'].apply(lambda x: x**2)
```

The `.apply(lambda x: x**2)` part is used to apply a function to each element in the selected 'sepal*length' column. In this case, a lambda function is used. The lambda function takes an input parameter x (each individual sepal*length value) and squares it.

# Summary: Data Cleaning and Pre-processing

Objective: Focus on cleaning and preprocessing extracted data for analysis.

Practical Task: Use pandas for tasks like handling missing values, removing duplicates, and transforming data types. Discuss the importance of data quality.

The most common mistake with spreadsheets is that we treat them like lab notebooks, that is relying on context, side notes, margins, spacial layouts and fields and metadata. We can usually interpret thiese things but computers don't view information in the same way. We need to explain to the computer what each step means. It can't firgure out how data is suppose to fit together.

What is data cleaning and preprocessing. Making it computer ready so it can analyzed.

What are common things we need to look out for:

1. Not filling in zeros - different to blank, a zero is actual data that was measured
2. Null Values - different to zero, null was not measured and thus should be ignored
3. Formatting to make data sheet pretty - highlighting and similar - add a new column instead with info
4. Comments in cells - place in separate column
5. Entering more than one piece of information in a cell - only one piece of information per cell
6. Using problematic field names - avoid spaces, numbers, and special characeters
7. Using special characters in data - avoid in your data

# Load

It is always useful to export the data after you have cleaned and performed the transformations. This can be done various formats. Note, we are outputting the files to a new folder, which we will need to create first.

## CSV

```
df.to_csv("data_02/output/iris_data_cleaned.csv")
```

If you don't want the Pandas index column you can specify:

```
df.to_csv("data_02/output/iris_data_cleaned.csv", index=False)
```

## Excel

```
df.to_excel("data_02/output/iris_data_cleaned.xlsx", index=False, sheet_name='Sheet1')
```

## JSON

```
df.to_json("data_02/output/iris_data_cleaned.json", orient='records')
```

In JSON format, there isn't a concept of a DataFrame index like in tabular data. The orient='records' argument specifies that the JSON file should be structured as a list of records, and the DataFrame index is not considered.

## Advanced Applications

For advanced users you can learn how to automate ETL processes for efficiency. You can use tools like Apache Airflow or create simple Python scripts to automate ETL pipelines. Schedule tasks to run at specified intervals. You can also use logging to track the progress of an ETL pipeline

and handle exceptions.

---

Well done if you have gotten this far. Remember to try all these examples yourself.

> *Start thinking of how you can use your own data in Python*