10 Working with Data

Today, we're living in a data-driven world with the availability of vast and increasing quantities of data in all sectors of life. Analysing this data is a fundamental key to effectively understanding it and identifying trends, patterns, or even abnormalities in its behaviour. This helps in interpreting important insights and making the right decisions.

Data analysis has a long list of tools that help scientists and engineers get a sense of the data they have and build on it. Python is one of the most common tools most academics use to easily load, understand, clean, interpret, and visualise the dynamics of the data. In the following, we will cover, with examples, the main aspects of **pandas** which is an open-source library providing high-performance, easy-to-use data structures and data analysis tools for Python.

10.1 Pandas & Dataframes

We already have covered the library **numpy** and how it is useful to store data in a rectangular structure. But there is a downside to this: you can only have data of the same type in there. In practice, you'll be working with data of different types: numerical values, strings, booleans, and so on. To easily and efficiently handle this data, we use Python Data Analysis Library **pandas** which is a high-level data manipulation tool mainly used for data analysis. **pandas** allows importing data from various file formats such as comma-separated values, JSON, SQL, and Microsoft Excel. **pandas** allows various data manipulation operations such as merging, reshaping, selecting, as well as data cleaning. In **pandas**, we store data in a so-called dataframe. A dataframe is a two-dimensional labeled tabular data structure that contains rows that represent "observations or samples" and columns that represent "attributes or features". To clarify this, let's have a look at Fig(48) which shows a built-in dataframe called "iris". This dataset consists of 3 different types of irises' (Setosa, Versicolour, and Virginica) petal and sepal lengths and widths, stored in a $m \times n = 150 \times 4$ table, where m and n denote the number of rows (samples) and columns (features), respectively. Each row has a unique row label: $0, 1, 2, ...$ that refers to a certain entry (or sample or observation), and each column is identified by its column label: Sepal Length, Sepal Width, Petal Length and Petal Width, where each one represents a feature (or an attribute) of this dataset.

10.2 Importing Datasets

The first step in any data analysis endeavour is to get the raw data into your program. Here, we will look at methods of loading data from two sources: a CSV

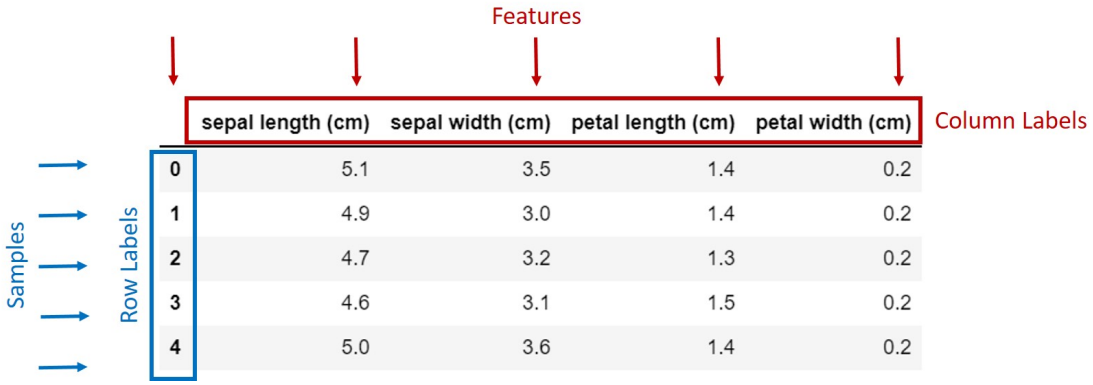


Figure 48: Dataframe Structure. Example: Iris Dataframe.

and an Excel file.

10.2.1 Loading a CSV file

Comma-separated values (CSV) files with `.csv` extension represent plain text files that contain records of data with comma-separated values. The CSV format is primarily used for storing tabular data, i.e., data that can be decomposed into rows and columns. Each row in a CSV file is a new record from the set of records contained in the file that, in turn, can be made up of one or more fields. Files with CSV format have several advantages [8]:

- CSV is easy to create. In fact, it can be done using any text editor.
- CSV format is human-readable, i.e., the data is not encoded or converted to binary before storing. This increases the readability as well as the ease of manipulation. Moreover, it is easy to edit.
- CSV files can be read using almost any text editor.

In Python, importing a comma-separated values (CSV) file has a specific syntax to follow. As shown in Fig(49), the code should start first by importing **pandas** library. Next, to import and read the CSV file, `read` method should be called from **pandas** library. In the Figure, a dataset movies.csv is imported and read by Python. You can surely take a glance at how this dataset is structured by calling the `head()` method that shows you the first five rows of the dataset, or you can call the `shape` method that shows you how many rows (samples) and columns (features) the dataset has. In this particular example, the dataset has 27278 movies and 3 features: movieID, title, and genres.

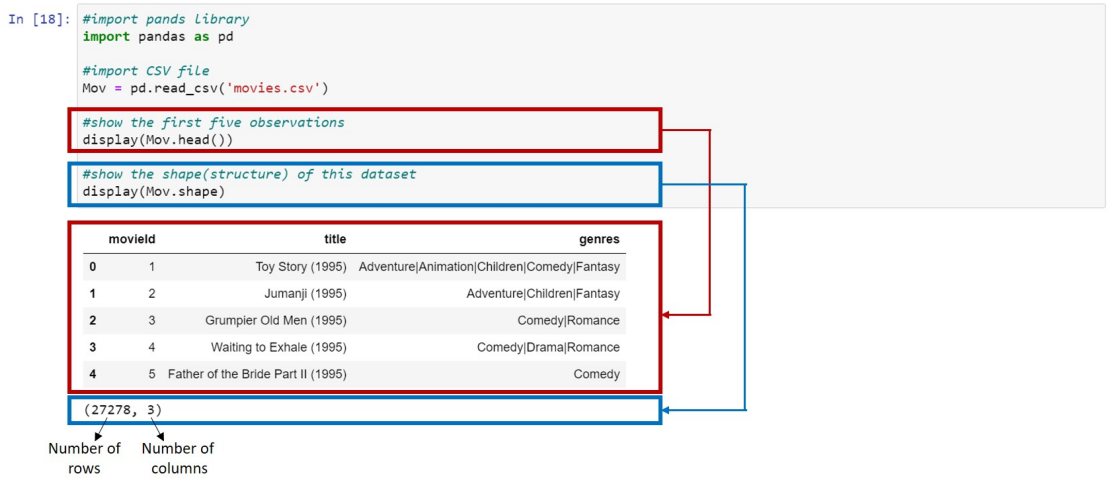


Figure 49: Python Syntax for importing a CSV file.

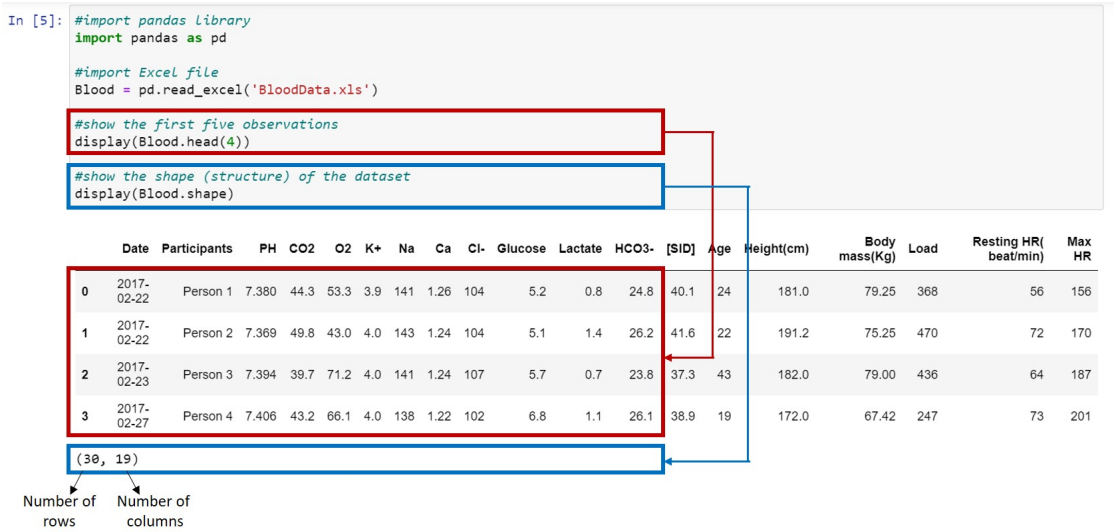
10.2.2 Loading an Excel file

Another type of format that can be used to store the tabular structure of data in Excel files. Excel files with `.xlsx` extension allow people to handle and store data in a controlled, easy, versatile, low-cost, and simple manner. Excel's popularity and continuous expansion are due to the fact that Excel handles data in a flexible way and is compatible with several data file formats, i.e., it exports data to `.txt`, `.csv`, or `.xls` formats.

An Excel file can be loaded in a similar way to a csv file. To import and read the Excel file, `read` method should be called from **pandas** library. As shown in Fig(50), a dataset of 19 blood chemical properties (number of columns) taken from 30 participants in their rest condition (number of rows) is loaded and read by Python.

10.3 Understanding & Manipulating Data

Quite possibly the most important part of data analysis is understanding the data you are working with and how it relates to the task you want to solve. It will not be effective to randomly choose an algorithm and throw your data at it. It is necessary to understand what is going on in your dataset before you begin to analyse it, e.g., inspect for any abnormalities and peculiarities. In the following, we will cover a couple of main tools that are commonly used to clean and pre-process datasets.



	Date	Participants	PH	CO2	O2	K+	Na	Ca	Cl-	Glucose	Lactate	HCO3-	[SID]	Age	Height(cm)	Body mass(Kg)	Load	Resting HR(beat/min)	Max HR
0	2017-02-22	Person 1	7.380	44.3	53.3	3.9	141	1.26	104	5.2	0.8	24.8	40.1	24	181.0	79.25	368	56	156
1	2017-02-22	Person 2	7.369	49.8	43.0	4.0	143	1.24	104	5.1	1.4	26.2	41.6	22	191.2	75.25	470	72	170
2	2017-02-23	Person 3	7.394	39.7	71.2	4.0	141	1.24	107	5.7	0.7	23.8	37.3	43	182.0	79.00	436	64	187
3	2017-02-27	Person 4	7.406	43.2	66.1	4.0	138	1.22	102	6.8	1.1	26.1	38.9	19	172.0	67.42	247	73	201

Figure 50: Python Syntax for importing an Excel file.

10.3.1 Data Inspection

After we load the dataset, it is a good idea to understand how it is structured and what kind of information it contains. Ideally, we would view the full data directly. But with most real-world cases, the data could have thousands to hundreds of thousands to millions of rows and columns. Instead, we have to rely on pulling samples to view small slices and calculating summary statistics of the data [5]. In Python, there are a couple of methods that are usually used to initially inspect the structure and the information of the dataset being analysed. We will go through them using the `Blood` dataset mentioned before as a toy example. First and foremost, it's very helpful to know the dimensionality (or the structure) of your dataset, i.e., how many rows (observations) and how many columns (features). This, as shown in Fig(51), can be done using the `shape` method. It displays two numbers, the first represents the number of rows, and the second represents the number of columns.

After knowing how many rows and columns there are in the dataset, it is quite helpful to have a look at the dataset itself. There are two methods to use in this case: `head()` and `tail()`. As shown in Fig(52), the former prints the first five observations and the latter prints the last five observations. Worth noting here that the default number of elements to display is five, but you may pass a custom number.

Another useful method to mention is `describe()`. This method is used to compute and view some basic statistical details like percentile, mean, median, standard deviation, minimum, maximum, etc. of a dataframe or a series of numeric

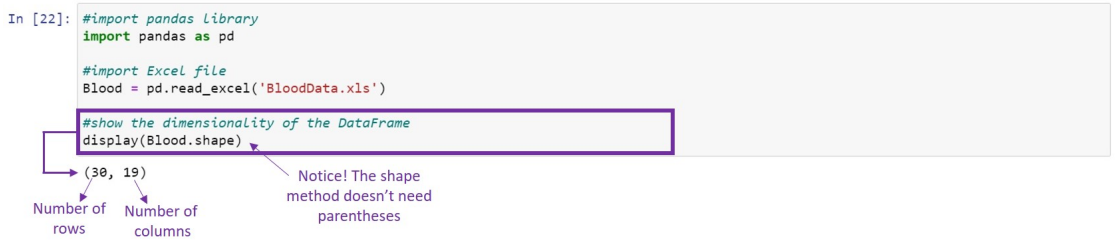
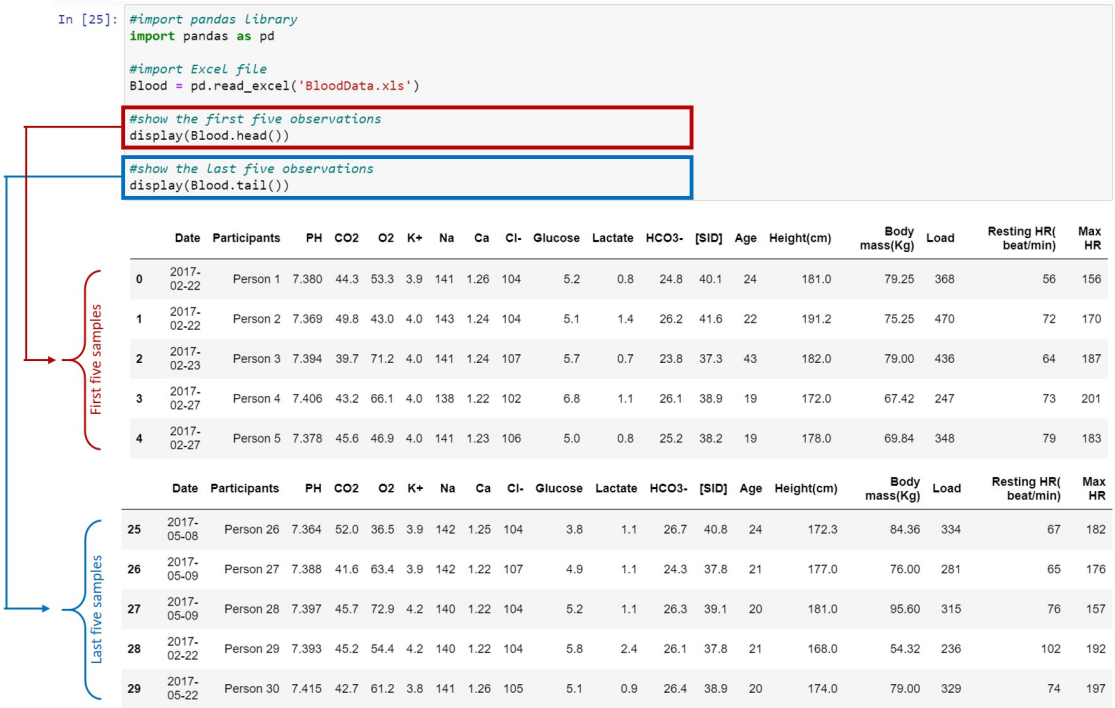


Figure 51: The `shape` method to print the structure of the dataframe.



	Date	Participants	PH	CO2	O2	K+	Na	Ca	Cl-	Glucose	Lactate	HCO3-	[SID]	Age	Height(cm)	Body mass(Kg)	Load	Resting HR(beat/min)	Max HR
0	2017-02-22	Person 1	7.380	44.3	53.3	3.9	141	1.26	104	5.2	0.8	24.8	40.1	24	181.0	79.25	368	56	156
1	2017-02-22	Person 2	7.369	49.8	43.0	4.0	143	1.24	104	5.1	1.4	26.2	41.6	22	191.2	75.25	470	72	170
2	2017-02-23	Person 3	7.394	39.7	71.2	4.0	141	1.24	107	5.7	0.7	23.8	37.3	43	182.0	79.00	436	64	187
3	2017-02-27	Person 4	7.406	43.2	66.1	4.0	138	1.22	102	6.8	1.1	26.1	38.9	19	172.0	67.42	247	73	201
4	2017-02-27	Person 5	7.378	45.6	46.9	4.0	141	1.23	106	5.0	0.8	25.2	38.2	19	178.0	69.84	348	79	183

	Date	Participants	PH	CO2	O2	K+	Na	Ca	Cl-	Glucose	Lactate	HCO3-	[SID]	Age	Height(cm)	Body mass(Kg)	Load	Resting HR(beat/min)	Max HR
25	2017-05-08	Person 26	7.364	52.0	36.5	3.9	142	1.25	104	3.8	1.1	26.7	40.8	24	172.3	84.36	334	67	182
26	2017-05-09	Person 27	7.388	41.6	63.4	3.9	142	1.22	107	4.9	1.1	24.3	37.8	21	177.0	76.00	281	65	176
27	2017-05-09	Person 28	7.397	45.7	72.9	4.2	140	1.22	104	5.2	1.1	26.3	39.1	20	181.0	95.60	315	76	157
28	2017-02-22	Person 29	7.393	45.2	54.4	4.2	140	1.22	104	5.8	2.4	26.1	37.8	21	168.0	54.32	236	102	192
29	2017-05-22	Person 30	7.415	42.7	61.2	3.8	141	1.26	105	5.1	0.9	26.4	38.9	20	174.0	79.00	329	74	197

Figure 52: To display the first or last five rows of a dataframe, two methods can be applied: `head()` and `tail()`, respectively.

values, as shown in Fig(53).

The last method to mention is `info()`. This method allows you to have a quick overview of your dataframe contents. It comes really handy when doing exploratory analysis of the data. As shown in Fig(54), it is used to print a concise summary of the dataframe including column data types, non-null values, and memory usage.

```
In [23]: #import pandas library
         import pandas as pd

         #import Excel file
         Blood = pd.read_excel('BloodData.xls')

         #Generate descriptive statistics
         display(Blood.describe())
```

	PH	CO2	O2	K+	Na	Ca	Cl-	Glucose	Lactate	HCO3-	[SID]	Age	Height(cm)	mas
count	30.000000	30.000000	30.000000	30.000000	30.000000	30.000000	30.000000	30.000000	30.000000	30.000000	30.000000	30.000000	30.000000	30.0
mean	7.390433	44.930000	55.473333	3.973333	141.466667	1.225667	104.533333	5.223333	1.106667	25.676667	39.800000	24.066667	178.293333	76.2
std	0.023322	4.135561	14.467799	0.204995	1.357821	0.030021	1.382984	0.681116	0.419304	1.074367	1.909865	6.781991	5.478795	8.5
min	7.341000	36.300000	31.300000	3.600000	138.000000	1.170000	102.000000	3.400000	0.400000	22.900000	35.900000	19.000000	168.000000	54.3
25%	7.373750	42.725000	47.000000	3.825000	141.000000	1.212500	104.000000	5.000000	0.825000	25.125000	38.550000	20.000000	175.700000	70.8
50%	7.391000	44.650000	52.800000	3.950000	142.000000	1.230000	104.000000	5.200000	1.100000	25.750000	39.550000	21.000000	178.000000	76.0
75%	7.403750	47.975000	69.250000	4.100000	142.750000	1.247500	105.000000	5.625000	1.275000	26.200000	41.400000	24.000000	181.750000	80.0
max	7.442000	52.100000	80.100000	4.500000	143.000000	1.290000	107.000000	6.800000	2.400000	29.000000	43.600000	43.000000	191.200000	95.6

Figure 53: The `describe()` method is used to show the main statistical information of the dataframe.

```
In [24]: #import pandas library
         import pandas as pd

         #import Excel file
         Blood = pd.read_excel('BloodData.xls')

         #Print a concise summary of a DataFrame
         display(Blood.info())
```

```
<class 'pandas.core.frame.DataFrame'>          ⟶ Type of dataset
RangeIndex: 30 entries, 0 to 29                ⟶ Structure of DataFrame
Data columns (total 19 columns):               ⟶ Columns Information
 #   Column      Missing values ⟵ Non-Null Count  Dtype   ⟶ Data type
---  ------                     --------------  -----
 0       Date                   30 non-null     datetime64[ns]
 1   Participants               30 non-null     object
 2   PH                         30 non-null     float64
 3   CO2                        30 non-null     float64
 4   O2                         30 non-null     float64
 5   K+                         30 non-null     float64
 6   Na                         30 non-null     int64
 7   Ca                         30 non-null     float64
 8   Cl-                        30 non-null     int64
 9   Glucose                    30 non-null     float64
 10  Lactate                    30 non-null     float64
 11  HCO3-                      30 non-null     float64
 12  [SID]                      30 non-null     float64
 13        Age                  30 non-null     int64
 14    Height(cm)               30 non-null     float64
 15  Body mass(Kg)              30 non-null     float64
 16    Load                     30 non-null     int64
 17  Resting HR( beat/min)      30 non-null     int64
 18  Max HR                     30 non-null     int64
dtypes: datetime64[ns](1), float64(11), int64(6), object(1)
memory usage: 4.6+ KB                          ⟶ Memory Usage

None
```

Figure 54: The `info()` method is used to have a quick overview of the dataframe.

10.3.2 Slicing, Selecting, & Deleting Data

In data analysis, there is almost always a need to slice a dataset (chop the entire dataset into a sub-dataset), select individual rows and/or columns, or even delete rows and/or columns to get the right selection of data for the specific analysis in hand. In the following, we will cover the main syntaxes for slicing, selecting, and

deleting rows and columns taking the `Blood` dataset as a toy example.

Slicing All rows and columns in a **pandas** dataframe have a unique index value. By default, this index is an integer indicating the row/column position in the dataframe; however, it does not have to be. Dataframe indices can be set to be unique alphanumeric strings or customer numbers. To select individual rows/columns or slices of rows/columns, **pandas** provides two methods [8]:

- `loc`: is useful when the index of the dataframe is a label (e.g., a string).
- `iloc`: works by looking for the position in the dataframe. For example, `iloc[0]` will return the first row regardless of whether the index is an integer or a label.

As shown in Fig(55), to slice out a set of rows, we use either of the following syntaxes:

```
data.iloc[start row index:stop row index,:]
```

```
data.loc[start row label:stop row label,:]
```

When slicing in pandas the start bound is included in the output. The stop bound is one step beyond the row you want to select. The colon here brings all the columns in the sliced subset. On the other hand, to slice out a set of columns, we use similar syntaxes as before:

```
data.iloc[:,start column index:stop column index]
```

```
data.loc[:,start column label:stop column label]
```

Similarly here, the stop bound is one step beyond the column you want to select. The colon here brings all the rows in the sliced subset. Sometimes, you might be interested in slicing out both rows and columns of a given dataset. This can be done using one of the following syntaxes:

```
data.iloc[start row index:stop row index,start column index:stop column index]
data.loc[start row label:stop row label,start column label:stop column label]
```

In [33]:
```
#import pandas library
import pandas as pd

#import Excel file
Blood = pd.read_excel('BloodData.xls')

#Slicing Subsets of Rows
Blood.iloc[0:3,:] # Select rows 0, 1, 2 (row 3 is not selected) by index

#Slicing Subsets of Columns
Blood.iloc[:,4:7] # Select columns 4,5,6 (column 7 is not selected) by index
#Blood.loc[:,"O2":"Na"] #by label

#Slicing Subsets of Rows and Columns
Blood.iloc[0:3,4:7] #Select rows 0, 1, 2 (row 3 is not selected) and columns 4,5,6 (column 7 is not selected) by index
#Blood.loc[0:3,"O2":"Na"] #by label
```

Figure 55: Slicing subsets of rows, columns, and both.

Selecting Instead of slicing out the dataset, vertically or horizontally, you might be interested in only certain rows or columns. In this case, as shown in Fig(56), to select a particular number of rows and columns, the same methods of `iloc` and `loc` apply here by specifying their indices or the labels, respectively. Also, to select a single value from the dataset, the following syntax is used: `data.iloc[a,b]` where `a` is the index of the row and `b` is the index of the column where the element lies in.

In [44]:
```
#import pandas library
import pandas as pd

#import Excel file
Blood = pd.read_excel('BloodData.xls')

#select one row
Blood.iloc[5,:] #by index

#select multiple rows
Blood.iloc[[5,7],:] #by index

#select one column
Blood.iloc[:,3] #by index
#Blood.loc[:,"CO2"] #by label

#select multiple columns
Blood.iloc[:,[3,8]] #by index
#Blood.loc[:,["CO2","O2"]] #by label

#Select list of rows and columns.
Blood.iloc[[1,6,8],[0,6,9]] #by index

#extract the data at the specified row and column location.
Blood.iloc[0,3] #show the value resides in the row with index 0 and column with index 3
```

Figure 56: Selecting rows, columns, and both.

Sometimes, you need to access the last row (or column) in your dataframe. Interestingly, this can be done in Python using the `[-1]` index. Fig(57) shows how to extract the last row, last column, and the last element from the last row and last column.

Deleting Also, sometimes you are interested in deleting one or more columns or rows out of the dataset you have. This can be easily done using the `drop` method as shown in Fig(58). To delete one or multiple rows, the process is easily done

```
In [4]: #import pandas library
        import pandas as pd

        #import Excel file
        Blood = pd.read_excel('BloodData.xls')

        #Select last row
        Blood.iloc[[-1],:]

        #Select last column
        Blood.iloc[:,[-1]]

        #Select last element from the last row and last column
        Blood.iloc[[-1],[-1]]
```

Figure 57: Selecting the last row, last column, and the last element from the last row and last column.

by identifying the index(ies) of the row(s) being deleted. However, to delete one or more columns the process is slightly different. The default way to use `drop` to remove columns is to provide the column indices (or labels) to be deleted along with specifying the `axis` parameter to be `1`.

```
In [58]: #import pandas library
         import pandas as pd

         #import Excel file
         Blood = pd.read_excel('BloodData.xls')

         #delete one row
         Blood.drop([0]) #by index

         #delete multiple rows
         Blood.drop([0,6]) #by index

         #delete one column
         Blood.drop(Blood.columns[0], axis=1) #delete the first column by index
         #Blood.drop('Participants', axis=1) #by label

         #delete multiple columns
         Blood.drop(Blood.columns[[0,1,3]], axis=1) #delete the first, second and fourth columns by index
         #Blood.drop(['PH','O2'], axis=1) #by label
```

Figure 58: Deleting rows and columns from a dataset.

10.3.3 Handling Missing Values

Real-world data would certainly have missing values. This could be due to many reasons such as data entry errors or data collection problems. Irrespective of the reasons, it is important to handle missing data because any statistical results based on a dataset with non-random missing values could be biased. Oftentimes, a dataset either shows an empty cell or uses a specific value to denote a missing observation, such as `NONE`, `-999`, `n/a`, `NA`, or `NaN`. To transform the dataset into a clean and organised format ready to be used for data analysis tasks, these missing values should be dealt with properly. There is no generic way to follow when it comes to missing values. The choice of how to deal with them depends entirely on the nature of the dataset being analysed. Having said that, there are some com-

mon practices that are usually used to tackle such values. For example, dropping columns with missing values, filling in the missing values with some number, such as zero, or the mean of the row or column the value lies in. In Python, you can identify whether a certain column has any missing values by using the `isnull()` method, as shown in Fig(59). The output is a boolean parameter that indicates whether there are (`True`) or aren't (`False`) any missing values in each row of that column.

```
In [67]: #import pandas library
         import pandas as pd

         #import Excel file
         Blood = pd.read_excel('BloodData.xls')

         #Check for any missing values in a specific column
         Blood['Participants'].isnull()
```

```
Out[67]: 0     False
         1     False
         2     False
         3     False
         4     False
         5     False
         6     False
         7     False
         8     False
         9     False
         10    False
         11    False
         12    False
         13    False
         14    False
         15    False
         16    False
         17    False
         18    False
         19    False
         20    False
         21    False
```

Figure 59: `isnull()` method to check whether the column "Participants" has any missing values.

Another useful way is to see the total number of missing values for each feature in the dataset. This can be done by two methods `isnull()` and `sum()`, as shown in Fig(60).

```
In [68]: #import pandas library
         import pandas as pd

         #import Excel file
         Blood = pd.read_excel('BloodData.xls')

         #Check for any missing values in a specific column
         Blood['Participants'].isnull()

         #Show the total missing values for each feature
         Blood.isnull().sum()

         #drop all rows that have missing data
         #Blood.dropna(how="all", inplace=True)
```

Figure 60: `isnull()` and `sum()` methods are used to show the total number of missing values in each column.