**Data Pre-processing using Python**

**Major Tasks in Data Pre-processing:**,,

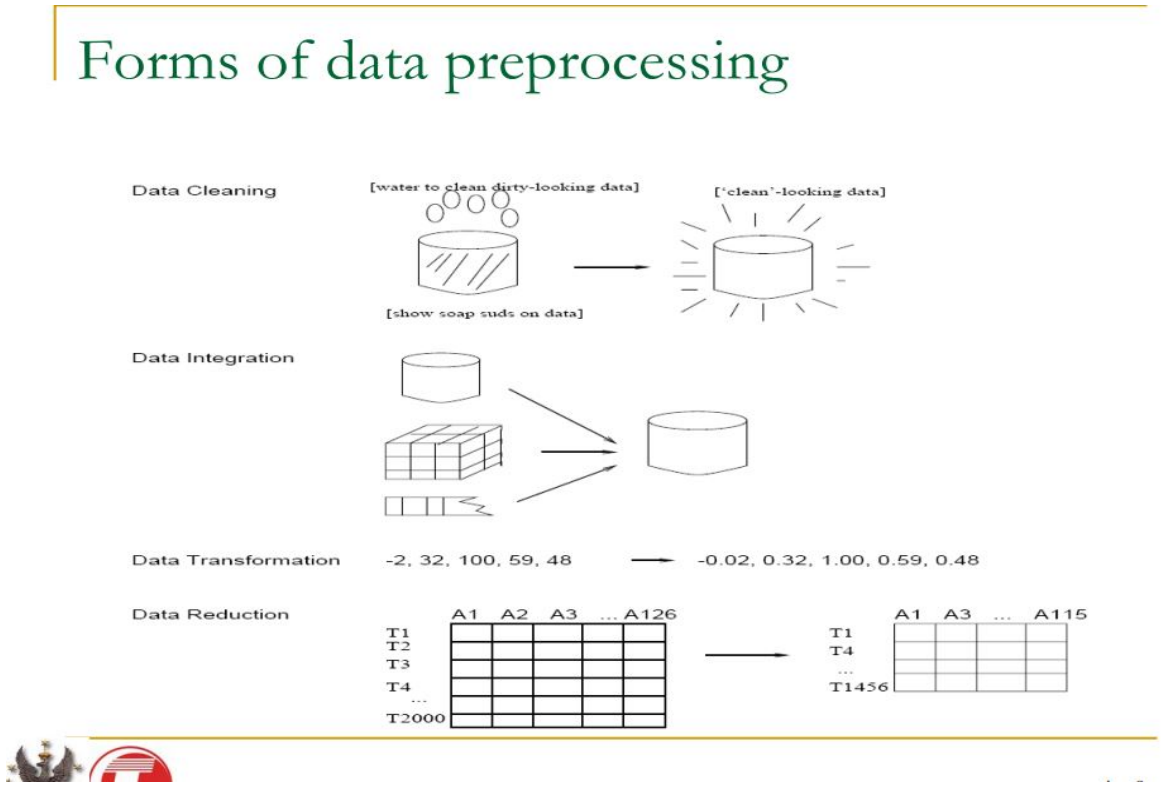
**Data cleaning**: Fill in missing values, smooth noisy data, identify or remove outliers, and resolve inconsistencies.

**Data integration:** Integration of multiple databases, data cubes, or files.

**Data transformation**: Normalization and aggregation.

**Data reduction**: Obtains reduced representation in volume but produces the same or similar analytical results.

**Data discretization**: Part of data reduction but with particular importance, especially for numerical data.




**Perform Data Pre processing**

**Learning Resources:**

1. https://www.youtube.com/watch?v=EaGbS7eWSs0

2. https://www.youtube.com/watch?v=ePHZlT1XjrA

**Pandas**

Python pandas is an excellent software library for manipulating data and analyzing it. It will let us manipulate numerical tables and time series using data structures and operations.

**Numpy**

Python numpy is another library we will use here. It lets us handle arrays and matrices, especially those multidimensional. It also provides several high-level mathematical functions to help us operate on these.

1. **Python Data Cleansing**

When some part of our data is missing, due to whichever reason, the accuracy of our predictions plummets.
Python Pandas will depict a missing value as NaN, which is short for Not a Number. Simply using the reindex() method will fill in NaN for blank values.

```
frame=pd.DataFrame(np.random.randn(4,3),index=[1,2,4,7],columns=['A','B','C'])
frame.reindex([1,2,3,4,5,6,7])
```

## a. Finding which columns have missing values

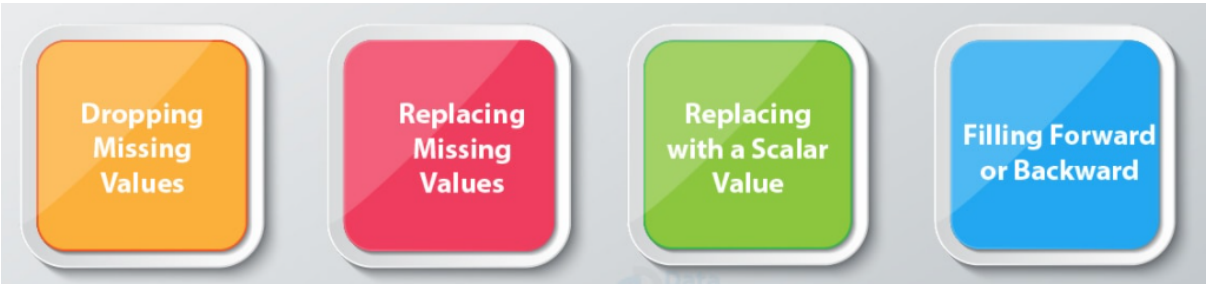
In the tutorial on wrangling, we saw how to find out which columns have missing values-

```
1.  >>> frame=frame.reindex([1,2,3,4,5,6,7])
2.  >>> frame['B'].isnull()
```

```
frame=frame.reindex([1,2,3,4,5,6,7])
frame['B'].isnull()
```

**Ways to Cleanse Missing Data in Python**

To perform a Python data cleansing, you can drop the missing values, replace them, replace each NaN with a scalar value, or fill forward or backward.



**a. Dropping Missing Values**

You can exclude missing values from your dataset using the dropna() method.

```
frame.dropna()
```

**b. Replacing Missing Values**

To replace each NaN we have in the dataset, we can use the replace() method.

```
from numpy import NaN

frame.replace({NaN:0.00})
```

**c. Replacing with a Scalar Value**

We can use the fillna() method for this.

```
frame.fillna(7)
```

**d. Filling Forward or Backward**

If we supply a *method* parameter to the fillna() method, we can fill forward or backward as we need. To fill forward, use the methods *pad* or *fill*, and to fill backward, use *bfill* and *backfill*.

```
frame.fillna(method='pad')
```

2. **Data Integration**

So far, we've made sure to remove the impurities in data and make it clean. Now, the next step is to combine data from different sources to get a unified structure with more meaningful and valuable information. This is mostly used if the data is segregated into different sources. To make it simple, let's assume we have data in CSV format in different places, all talking about the same scenario. Say we have some data about an employee in a database. We can't expect all the data about the employee to reside in the same table. It's possible that the employee's personal data will be located in one table, the employee's project history will be in a second table, the employee's time-in and time-out details will be in another table, and so on. So, if we want to do some analysis about the employee, we need to get all the employee data in one common place. This process of bringing data together in one place is called data integration. To do data integration, we can merge multiple pandas DataFrames using the **merge** function.

we'll merge the details of students from two datasets, namely **student.csv** and **marks.csv**. The **student** dataset contains columns such as **Age**, **Gender**, **Grade**, and **Employed**.

The **marks.csv** dataset contains columns such as **Mark** and **City**. The **Student_id** column is common between the two datasets. Follow these steps

The **student.csv** dataset
location: https://github.com/TrainingByPackt/Data-Science-with-Python/blob/master/Chapter01/Data/student.csv.

The **marks.csv** dataset
location: https://github.com/TrainingByPackt/Data-Science-with-Python/blob/master/Chapter01/Data/mark.csv.

```
import pandas as pd

from google.colab import files
data = files.upload()

#Reading data from csv file
my_data1 = pd.read_csv("Student.csv")
print(my_data1.head())
my_data2 = pd.read_csv("mark.csv")
print(my_data2.head())
df = pd.merge(my_data1, my_data2, on = 'Student_id')
df.head(10)
```

3. **Data Transformation**

We have a lot of columns that have different types of data. Our goal is to transform the data into a machine-learning-digestible format. All machine learning algorithms are based on mathematics. So, we need to convert all the columns into numerical format.

Taking a broader perspective, data is classified into numerical and categorical data:

**Numerical**: As the name suggests, this is numeric data that is quantifiable.

**Categorical**: The data is a string or non-numeric data that is qualitative in nature.

**Handling Categorical Data**

There are some algorithms that can work well with categorical data, such as decision trees. But most machine learning algorithms cannot operate directly with categorical data. These algorithms require the input and output both to be in numerical form. If the output to be predicted is categorical, then after prediction we convert them back to categorical data from numerical data. Let's discuss some key challenges that we face while dealing with categorical data:

## Encoding

To address the problems associated with categorical data, we can use encoding. This is the process by which we convert a categorical variable into a numerical form. Here, we will look at three simple methods of encoding categorical data.

## Replacing

This is a technique in which we replace the categorical data with a number. This is a simple replacement and does not involve much logical processing. Let's look at an exercise to get a better idea of this.

### Simple Replacement of Categorical Data with a Number

In this exercise, we will use the **student** dataset that we saw earlier. We will load the data into a pandas dataframe and simply replace all the categorical data with numbers. Follow these steps to complete this exercise:

The **student** dataset can be found at this location: https://github.com/TrainingByPackt/Data-Science-with-Python/blob/master/Chapter01/Data/student.csv.

1. Open a Jupyter notebook and add a new cell. Write the following code to import pandas and then load the dataset into the pandas dataframe:

```
import pandas as pd
import numpy as np
dataset = "https://github.com/TrainingByPackt/Data-Science-with-python/blob/master/Chapter01/Data/student.csv"
df = pd.read_csv(dataset, header = 0)
```

2. Find the categorical column and separate it out with a different dataframe. To do so, use the **select_dtypes()** function from pandas:

```
df_categorical = df.select_dtypes(exclude=[np.number])
df_categorical
```

The preceding code generates the following output:

| | Gender | Grade | Employed |
|---|---|---|---|
| 0 | Male | 1st Class | yes |
| 1 | Female | 2nd Class | no |
| 2 | Male | 1st Class | no |
| 3 | Female | 2nd Class | no |
| 4 | Male | 1st Class | no |

*Figure 1.28: Categorical columns of the dataframe*

1. Find the distinct unique values in the **Grade** column. To do so, use the **unique()** function from pandas with the column name:

   df_categorical['Grade'].unique()

   The preceding code generates the following output:

```
array(['1st Class', '2nd Class', '3rd Class'], dtype=object)
```

*Figure 1.29: Unique values in the Grade column*

2. Find the frequency distribution of each categorical column. To do so, use the **value_counts()** function on each column. This function returns the counts of unique values in an object:

   df_categorical.Grade.value_counts()

   The output of this step is as follows:

```
2nd Class    80
3rd Class    80
1st Class    72
Name: Grade, dtype: int64
```

*Figure 1.30: Total count of each unique value in the Grade column*

1. Replace the entries in the **Grade** column. Replace **1st class** with **1**, **2nd class** with **2**, and **3rd class** with **3**. To do so, use the **replace()** function:

   df_categorical.Grade.replace({"1st Class":1, "2nd Class":2, "3rd Class":3}, inplace= True)

2. Replace the entries in the **Gender** column. Replace **Male** with **0** and **Female** with **1**. To do so, use the **replace()** function:

   df_categorical.Gender.replace({"Male":0,"Female":1}, inplace= True)

3. Replace the entries in the **Employed** column. Replace **no** with **0** and **yes** with **1**. To do so, use the **replace()** function:

   df_categorical.Employed.replace({"yes":1,"no":0}, inplace = True)

4. Once all the replacements for three columns are done, we need to print the dataframe. Add the following code:
   df_categorical.head()

You have successfully converted the categorical data to numerical data using a simple manual replacement method. We will now move on to look at another method of encoding categorical data.

Source:
https://subscription.packtpub.com/book/data/9781838552862/1/ch01lvl1sec07/data-transformation

**4.Data Reduction**

a. **Normaliztion:**

**Normalization** refers to rescaling real-valued numeric attributes into a 00 to 11 range.

**Data normalization** is used in machine learning to make model training less sensitive to the scale of features. This allows our model to converge to better weights and, in turn, leads to a more accurate model.

Normalization makes the features more consistent with each other, which allows the model to predict outputs more accurately.

**Normalization** is used to scale the data of an attribute so that it falls in a smaller range, such as -1.0 to 1.0 or 0.0 to 1.0. It is generally useful for classification algorithms.

**Need of Normalization –**

Normalization is generally required when we are dealing with attributes on a different scale, otherwise, it may lead to a dilution in effectiveness of an important equally important attribute(on lower scale) because of other attribute having values on larger scale.
In simple words, when multiple attributes are there but attributes have values on different scales, this may lead to poor data models while performing data mining operations. So they are normalized to bring all the attributes on the same scale.

## Code

Python provides the preprocessing library, which contains the normalize function to normalize the data. It takes an array in as an input and normalizes its values between 00 and 11. It then returns an output array with the same dimensions as the input.

```python
from sklearn import preprocessing
import numpy as np

a = np.random.random((1, 4))
a = a*20
print("Data = ", a)

# normalize the data attributes
normalized = preprocessing.normalize(a)
print("Normalized Data = ", normalized)
```

```
Data =  [[ 9.47187747 16.40319022 16.72005415  8.14876697]]
Normalized Data =  [[0.3567964  0.6178922  0.62982815 0.30695611]]
```