1ST EDITION

Exploratory Data Analysis with Python Cookbook

Over 50 recipes to analyze, visualize, and extract insights from structured and unstructured data

AYODELE OLULEYE

2
Preparing Data for EDA

Before exploring and analyzing tabular data, we sometimes will be required to prepare the data for analysis. This preparation can come in the form of data transformation, aggregation, or cleanup. In Python, the `pandas` library helps us to achieve this through several modules. The preparation steps for tabular data are never a one-size-fits-all approach. They are typically determined by the structure of our data, that is, the rows, columns, data types, and data values.

In this chapter, we will focus on common data preparation techniques required to prepare our data for EDA:

- Grouping data
- Appending data
- Concatenating data
- Merging data
- Sorting data
- Categorizing data
- Removing duplicate data
- Dropping data rows and columns
- Replacing data
- Changing a data format
- Dealing with missing values

Technical requirements

We will leverage the `pandas` library in Python for this chapter. The code and notebooks for this chapter are available on GitHub at `https://github.com/PacktPublishing/Exploratory-Data-Analysis-with-Python-Cookbook`.

Grouping data

When we group data, we are aggregating the data by category. This can be very useful especially when we need to get a high-level view of a detailed dataset. Typically, to group a dataset, we need to identify the column/category to group by, the column to aggregate by, and the specific aggregation to be done. The column/category to group by is usually a categorical column while the column to aggregate by is usually a numeric column. The aggregation to be done can be a count, sum, minimum, maximum, and so on. We can also perform aggregation such as count directly on the categorical column we group by

In `pandas`, the `groupby` method helps us group data.

Getting ready

We will work with one dataset in this chapter – the **Marketing Campaign** data from Kaggle.

Create a folder for this chapter and create a new Python script or Jupyter notebook file in that folder. Create a `data` subfolder and place the `marketing_campaign.csv` file in that subfolder. Alternatively, you can retrieve all the files from the GitHub repository.

> **Note**
>
> Kaggle provides the *Marketing Campaign* data for public use at `https://www.kaggle.com/datasets/imakash3011/customer-personality-analysis`. In this chapter, we use both the full dataset and samples of the dataset for the different recipes. The data is also available in the repository. The data in Kaggle appears in a single-column format, but the data in the repository was transformed into a multiple-column format for easy usage in pandas.

How to do it...

We will learn how to group data using the `pandas` library:

1. Import the `pandas` library:

```
import pandas as pd
```

2. Load the `.csv` file into a dataframe using `read_csv`. Then, subset the dataframe to include only relevant columns:

```
marketing_data = pd.read_csv("data/marketing_campaign.
csv")


marketing_data = marketing_data[['ID','Year_
Birth', 'Education','Marital_
Status','Income','Kidhome', 'Teenhome', 'Dt_
```

```
Customer',              'Recency','NumStorePurchases',
'NumWebVisitsMonth']]
```

3. Inspect the data. Check the first few rows and use `transpose` (`T`) to show more information. Also, check the data types as well as the number of columns and rows:

```
marketing_data.head(2).T
            0    1
ID    5524    2174
Year_Birth    1957    1954
Education    Graduation    Graduation
…        …        …
NumWebVisitsMonth    7    5

marketing_data.dtypes
ID    int64
Year_Birth    int64
Education    object
…           …
NumWebVisitsMonth    int64

marketing_data.shape
(2240, 11)
```

4. Use the `groupby` method in `pandas` to get the average number of store purchases of customers based on the number of kids at home:

```
marketing_data.groupby('Kidhome')['NumStorePurchases'].
mean()
Kidhome
0    7.2173240525908735
1    3.863181312569522
2    3.4375
```

That's all. Now, we have grouped our dataset.

How it works...

All of the recipes in this chapter use the `pandas` library for data transformation and manipulation. We refer to `pandas` as `pd` in *step 1*. In *step 2*, we use `read_csv` to load the `.csv` file into a `pandas` dataframe and call it `marketing_data`. We also subset the dataframe to include only 11 relevant columns. In *step 3*, we inspect the dataset using the `head` method to see the first two rows in the dataset; we also use `transform` (`T`) along with `head` to transform the rows into columns, due to the size of the data (i.e., it has many columns). We use the `dtypes` attribute of the dataframe to show the data types of all columns. Numeric data has `int` and `float` data types while character data has the `object` data type. We inspect the number of rows and columns using `shape`, which returns a tuple that displays the number of rows as the first element and the number of columns as the second element.

In *step 4*, we apply the `groupby` method to get the average number of store purchases of customers based on the number of kids at home. Using the `groupby` method, we group by `Kidhome`, then we aggregate by `NumStorePurchases`, and finally, we use the `mean` method as the specific aggregation to be performed on `NumStorePurchases`.

There's more...

Using the `groupby` method in `pandas`, we can group by multiple columns. Typically, these columns only need to be presented in a Python list to achieve this. Also, beyond the mean, several other aggregation methods can be applied, such as max, min, and median. In addition, the `agg` method can be used for aggregation; typically, we will need to provide specific `numpy` functions to be used. Custom functions for aggregation can be applied through the `apply` or `transform` method in pandas.

See also

Here is an insightful article by Dataquest on the `groupby` method in `pandas`: `https://www.dataquest.io/blog/grouping-data-a-step-by-step-tutorial-to-groupby-in-pandas/`.

Appending data

Sometimes, we may be analyzing multiple datasets that have a similar structure or samples of the same dataset. While analyzing our datasets, we may need to append them together into a new single dataset. When we append datasets, we stitch the datasets along the rows. For example, if we have 2 datasets containing 1,000 rows and 20 columns each, the appended data will contain 2,000 rows and 20 columns. The rows typically increase while the columns remain the same. The datasets are allowed to have a different number of rows but typically should have the same number of columns to avoid errors after appending.

In `pandas`, the `concat` method helps us append data.

Getting ready

We will continue working with the *Marketing Campaign* data from Kaggle. We will work with two samples of that dataset.

Place the `marketing_campaign_append1.csv` and `marketing_campaign_append2.csv` files in the data subfolder created in the first recipe. Alternatively, you could retrieve all the files from the GitHub repository.

How to do it...

We will explore how to append data using the `pandas` library:

1. Import the `pandas` library:

```
import pandas as pd
```

2. Load the `.csv` files into a dataframe using `read_csv`. Then, subset the dataframes to include only relevant columns:

```
marketing_sample1 = pd.read_csv("data/marketing_campaign_
append1.csv")
marketing_sample2 = pd.read_csv("data/marketing_campaign_
append2.csv")
marketing_sample1 = marketing_sample1[['ID', 'Year_
Birth','Education','Marital_Status','Income',
'Kidhome','Teenhome','Dt_Customer',
'Recency','NumStorePurchases', 'NumWebVisitsMonth']]


marketing_sample2 = marketing_sample2[['ID', 'Year_
Birth','Education','Marital_Status','Income',
'Kidhome','Teenhome','Dt_Customer',
'Recency','NumStorePurchases', 'NumWebVisitsMonth']]
```

3. Take a look at the two datasets. Check the first few rows and use `transpose` (`T`) to show more information:

```
marketing_sample1.head(2).T
    0    1
ID    5524    2174
Year_Birth    1957    1954
…         …         …
```

```
NumWebVisitsMonth    7    5

marketing_sample2.head(2).T
    0    1
ID    9135    466
Year_Birth    1950    1944
…        …        …
NumWebVisitsMonth    8    2
```

4. Check the data types as well as the number of columns and rows:

```
marketing_sample1.dtypes
ID    int64
Year_Birth    int64
…          …
NumWebVisitsMonth    int64

marketing_sample2.dtypes
ID    int64
Year_Birth    int64
…          …
NumWebVisitsMonth    int64

marketing_sample1.shape
(500, 11)
marketing_sample2.shape
(500, 11)
```

5. Append the datasets. Use the `concat` method from the `pandas` library to append the data:

```
appended_data = pd.concat([marketing_sample1, marketing_
sample2])
```

6. Inspect the shape of the result and the first few rows:

```
appended_data.head(2).T
    0    1
ID    5524    2174
```

```
Year_Birth    1957    1954
Education    Graduation    Graduation
Marital_Status    Single    Single
Income    58138.0    46344.0
Kidhome    0    1
Teenhome    0    1
Dt_Customer    04/09/2012    08/03/2014
Recency    58    38
NumStorePurchases    4    2
NumWebVisitsMonth    7    5

appended_data.shape
(1000, 11)
```

Well done! We have appended our datasets.

How it works...

We import the `pandas` library and refer to it as `pd` in *step 1*. In *step 2*, we use `read_csv` to load the two `.csv` files to be appended into `pandas` dataframes. We call the dataframes `marketing_sample1` and `marketing_sample2` respectively. We also subset the dataframes to include only 11 relevant columns. In *step 3*, we inspect the dataset using the `head` method to see the first two rows in the dataset; we also use transform (`T`) along with `head` to transform the rows into columns due to the size of the data (i.e., it has many columns). In *step 4*, we use the `dtypes` attribute of the dataframe to show the data types of all columns. Numeric data has int and float data types while character data has the object data type. We inspect the number of rows and columns using `shape`, which returns a tuple that displays the number of rows and columns respectively.

In *step 5*, we apply the `concat` method to append the two datasets. The method takes in the list of dataframes as an argument. The list is the only argument required because the default setting of the `concat` method is to append data. In *step 6*, we inspect the first few rows of the output and its shape.

There's more...

Using the `concat` method in `pandas`, we can append multiple datasets beyond just two. All that is required is to include these datasets in the list, and then they will be appended. It is important to note that the datasets must have the same columns.

Concatenating data

Sometimes, we may need to stitch multiple datasets or samples of the same dataset by columns and not rows. This is where we concatenate our data. While appending stitches rows of data together, concatenating stitches columns together to provide a single dataset. For example, if we have 2 datasets containing 1,000 rows and 20 columns each, the concatenated data will contain 1,000 rows and 40 columns. The columns typically increase while the rows remain the same. The datasets are allowed to have a different number of columns but typically should have the same number of rows to avoid errors after concatenating.

In `pandas`, the `concat` method helps us concatenate data.

Getting ready

We will continue working with the *Marketing Campaign* data from Kaggle. We will work with two samples of that dataset.

Place the `marketing_campaign_concat1.csv` and `marketing_campaign_concat2.csv` files in the data subfolder created in the first recipe. Alternatively, you can retrieve all the files from the GitHub repository.

How to do it...

We will explore how to concatenate data using the `pandas` library:

1. Import the `pandas` library:

```
import pandas as pd
```

2. Load the `.csv` files into a dataframe using `read_csv`:

```
marketing_sample1 = pd.read_csv("data/marketing_campaign_
concat1.csv")
marketing_sample2 = pd.read_csv("data/marketing_campaign_
concat2.csv")
```

3. Take a look at the two datasets. Check the first few rows and use `transpose` (`T`) to show more information:

```
marketing_sample1.head(2).T
    0    1
ID    5524    2174
Year_Birth    1957    1954
Education    Graduation    Graduation
```

```
Marital_Status    Single    Single
Income    58138.0    46344.0

marketing_sample2.head(2).T
    0    1
NumDealsPurchases    3    2
NumWebPurchases    8    1
NumCatalogPurchases    10    1
NumStorePurchases    4    2
NumWebVisitsMonth    7    5
```

4. Check the data types as well as the number of columns and rows:

```
marketing_sample1.dtypes
ID    int64
Year_Birth    int64
Education    object
Marital_Status    object
Income    float64

marketing_sample2.dtypes
NumDealsPurchases    int64
NumWebPurchases         int64
NumCatalogPurchases    int64
NumStorePurchases    int64
NumWebVisitsMonth     int64

marketing_sample1.shape
(2240, 5)
marketing_sample2.shape
(2240, 5)
```

5. Concatenate the datasets. Use the `concat` method from the `pandas` library to concatenate the data:

```
concatenated_data = pd.concat([marketing_sample1,
marketing_sample2], axis = 1)
```

6. Inspect the shape of the result and the first few rows:

```
concatenated_data.head(2).T
    0    1
ID    5524    2174
Year_Birth    1957    1954
Education    Graduation    Graduation
Marital_Status    Single    Single
Income    58138.0    46344.0
NumDealsPurchases    3    2
NumWebPurchases    8    1
NumCatalogPurchases    10    1
NumStorePurchases    4    2
NumWebVisitsMonth    7    5

concatenated_data.shape
(2240, 10)
```

Awesome! We have concatenated our datasets.

How it works...

We import the `pandas` library and refer to it as `pd` in *step 1*. In *step 2*, we use `read_csv` to load the two `.csv` files to be concatenated into `pandas` dataframes. We call the dataframes `marketing_sample1` and `marketing_sample2` respectively. In *step 3*, we inspect the dataset using `head(2)` to see the first two rows in the dataset; we also use transform (`T`) along with `head` to transform the rows into columns due to the size of the data (i.e., it has many columns). In *step 4*, we use the `dtypes` attribute of the dataframe to show the data types of all columns. Numeric data has int and float data types while character data has the object data type. We inspect the number of rows and columns using `shape`, which returns a tuple that displays the number of rows and columns respectively.

In *step 5*, we apply the `concat` method to concatenate the two datasets. Just like when appending, the method takes in the list of dataframes as an argument. However, it takes an additional argument for the axis parameter. The value `1` indicates that the axis refers to columns. The default value is typically `0`, which refers to rows and is relevant for appending datasets. In *step 6*, we check the first few rows of the output as well as the shape.

There's more...

Using the `concat` method in `pandas`, we can concatenate multiple datasets beyond just two. Just like appending, all that is required is to include these datasets in the list and the axis value, which is typically `1` for concatenation. It is important to note that the datasets must have the same number of rows.

See also

You can read this insightful article by Dataquest on concatenation: `https://www.dataquest.io/blog/pandas-concatenation-tutorial/`.

Merging data

Merging sounds a bit like concatenating our dataset; however, it is quite different. To merge datasets, we need to have a common field in both datasets on which we can perform a merge.

If you are familiar with the SQL or join commands, then you are probably familiar with merging data. Usually, data from relational databases will require merging operations. Relational databases typically contain tabular data and account for a significant proportion of data found in many organizations. Some key concepts to note when doing merge operations include the following:

- **Join key column**: This refers to the common column within both datasets in which there are matching values. This is typically used to join the datasets. The columns do not need to have the same name; they only need to have matching values within the two datasets.
- **Type of join**: There are different types of join operations that can be performed on datasets:
  - **Left join**: We retain all the rows in the left dataframe. Values in the right dataframe that do not match the values in the left dataframe are added as empty/**Not a Number** (**NaN**) values in the result. The matching is done based on the matching/join key column.
  - **Right join**: We retain all the rows in the right dataframe. Values in the left dataframe that do not match the values in the right dataframe are added as empty/NaN values in the result. The matching is done based on the matching/join key column.
  - **Inner join**: We retain only the common values in both the left and right dataframes in the result – that is, we do not return empty/NaN values.

- **Outer join/full outer join**: We retain all the rows for the left and right dataframes. If the values do not match, NaN is added to the result.

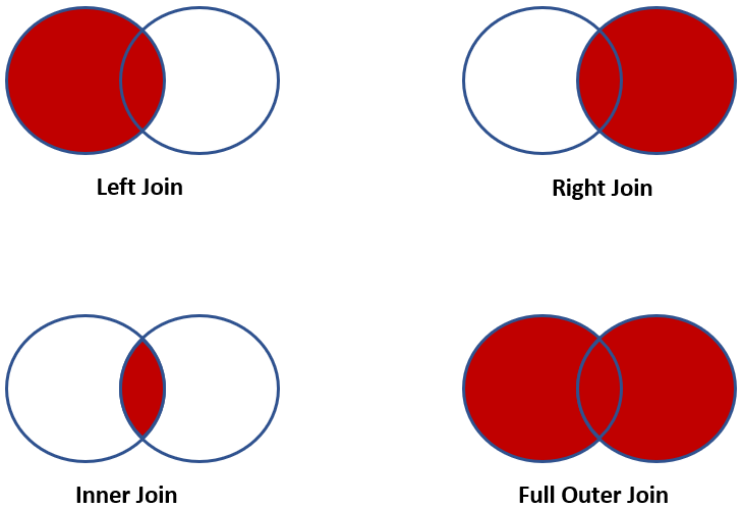

Figure 2.1 – Venn diagrams illustrating different types of joins

In `pandas`, the `merge` method helps us to merge dataframes.

Getting ready

We will continue working with the *Marketing Campaign* data from Kaggle. We will work with two samples of that dataset.

Place the `marketing_campaign_merge1.csv` and `marketing_campaign_merge2.csv` files in the data subfolder created in the first recipe. Alternatively, you can retrieve all the files from the GitHub repository.

How to do it...

We will merge datasets using the `pandas` library:

1. Import the `pandas` library:

```
import pandas as pd
```

2. Load the `.csv` files into a dataframe using `read_csv`:

```
marketing_sample1 = pd.read_csv("data/marketing_campaign_
merge1.csv")
```

```
marketing_sample2 = pd.read_csv("data/marketing_campaign_
merge2.csv")
```

3. Take a look at the two datasets. Check the first few rows through the `head` method. Also, check the number of columns and rows:

```
marketing_sample1.head()
      ID  Year_Birth  Education
0    5524  1957     Graduation
1    2174  1954     Graduation
2    4141  1965     Graduation
3    6182  1984     Graduation
4    5324  1981      PhD

    ID    Marital_Status    Income
0    5524    Single    58138.0
1    2174    Single    46344.0
2    4141    Together    71613.0
3    6182    Together    26646.0
4    5324    Married    58293.0

marketing_sample1.shape
(2240, 3)
marketing_sample2.shape
(2240, 3)
```

4. Merge the datasets. Use the `merge` method from the `pandas` library to merge the datasets:

```
merged_data = pd.merge(marketing_sample1,marketing_
sample2,on = "ID")
```

5. Inspect the shape of the result and the first few rows:

```
merged_data.head()
    ID    Year_Birth    Education    Marital_
Status    Income
0    5524    1957    Graduation    Single    58138.0
1    2174    1954    Graduation    Single    46344.0
2    4141    1965    Graduation    Together    71613.0
```

```
3    6182    1984    Graduation    Together    26646.0
4    5324    1981    PhD    Married    58293.0
merged_data.shape
(2240, 5)
```

Great! We have merged our dataset.

How it works...

We import the `pandas` library and refer to it as `pd` in *step 1*. In *step 2*, we use `read_csv` to load the two `.csv` files to be merged into `pandas` dataframes. We call the dataframes `marketing_sample1` and `marketing_sample2` respectively. In *step 3*, we inspect the dataset using `head()` to see the first five rows in the dataset. We inspect the number of rows and columns using `shape`, which returns a tuple that displays the number of rows and columns respectively.

In *step 4*, we apply the `merge` method to merge the two datasets. We provide four arguments for the `merge` method. The first two arguments are the dataframes we want to merge, the third specifies the key or common column upon which a merge can be achieved. The merge method also has a how parameter. This parameter specifies the type of join to be used. The default parameter of this argument is an inner join.

There's more...

Sometimes, the common field in two datasets may have a different name. The `merge` method allows us to address this through two arguments, `left_on` and `right_on`. `left_on` specifies the key on the left dataframe, while `right_on` is the same thing on the right dataframe.

See also

You can check out this useful resource by *Real Python* on merging data in `pandas`: `https://realpython.com/pandas-merge-join-and-concat/`.

Sorting data

When we sort data, we arrange it in a specific sequence. This specific sequence typically helps us to spot patterns very quickly. To sort a dataset, we usually must specify one or more columns to sort by and specify the order to sort by (ascending or descending order).

In `pandas`, the `sort_values` method can be used to sort a dataset.

Getting ready

We will work with the Marketing Campaign data (`https://www.kaggle.com/datasets/imakash3011/customer-personality-analysis`) for this recipe. Alternatively, you can retrieve this from the GitHub repository.

How to do it...

We will sort data using the `pandas` library:

1. Import the `pandas` library:

```
import pandas as pd
```

2. Load the `.csv` file into a dataframe using `read_csv`. Then, subset the dataframe to include only relevant columns:

```
marketing_data = pd.read_csv("data/marketing_campaign.
csv")


marketing_data = marketing_data[['ID','Year_
Birth', 'Education','Marital_
Status','Income','Kidhome', 'Teenhome', 'Dt_
Customer',              'Recency','NumStorePurchases',
'NumWebVisitsMonth']]
```

3. Inspect the data. Check the first few rows and use `transpose` (`T`) to show more information. Also, check the data types as well as the number of columns and rows:

```
marketing_data.head(2).T
                0    1
ID      5524    2174
Year_Birth      1957    1954
Education       Graduation      Graduation
…           …           …
NumWebVisitsMonth       7       5

marketing_data.dtypes
ID      int64
Year_Birth      int64
Education       object
…           …
```

```
NumWebVisitsMonth    int64

marketing_data.shape
(2240, 11)
```

4. Sort customers based on the number of store purchases in descending order:

```
sorted_data = marketing_data.sort_
values('NumStorePurchases', ascending=False)
```

5. Inspect the result. Subset for relevant columns:

```
sorted_data[['ID','NumStorePurchases']]
    ID    NumStorePurchases
1187    9855    13
803    9930    13
1144    819    13
286    10983    13
1150    1453    13
 ...     ...    ...
164    8475    0
2214    9303    0
27    5255    0
1042    10749    0
2132    11181    0
```

Great! We have sorted our dataset.

How it works...

We refer to `pandas` as `pd` in *step 1*. In *step 2*, we use `read_csv` to load the `.csv` file into a `pandas` dataframe and call it `marketing_data`. We also subset the dataframe to include only 11 relevant columns. In *step 3*, we inspect the dataset using `head(2)` to see the first two rows in the dataset; we also use `transpose` (`T`) along with `head` to transform the rows into columns due to the size of the data (i.e., it has many columns). We use the `dtypes` attribute of the dataframe to show the data types of all columns. Numeric data has int and float data types while character data has the object data type. We inspect the number of rows and columns using `shape`, which returns a tuple that displays the number of rows as the first element and the number of columns as the second element.

In *step 4*, we apply the `sort_values` method to sort the `NumStorePurchases` column. Using the `sort values` method, we sort `NumStorePurchases` in descending order. The method takes two arguments, the dataframe column to be sorted and the sorting order. `false` indicates a sort in descending order while `true` indicates a sort in ascending order.

There's more...

Sorting can be done across multiple columns in `pandas`. We can sort based on multiple columns by supplying columns as a list in the `sort_values` method. The sort will be performed in the order in which the columns are supplied – that is, column 1 first, then column 2 next, and subsequent columns. Also, a sort isn't limited to numerical columns alone; it can be used for columns containing characters.

Categorizing data

When we refer to categorizing data, we are specifically referring to binning, bucketing, or cutting a dataset. Binning involves grouping the numeric values in a dataset into smaller intervals called *bins* or *buckets*. When we bin numerical values, each bin becomes a categorical value. Bins are very useful because they can provide us with insights that may have been difficult to spot if we had worked directly with individual numerical values. Bins don't always have equal intervals; the creation of bins is dependent on our understanding of a dataset.

Binning can also be used to address outliers or reduce the effect of observation errors. Outliers are unusually high or unusually low data points that are far from other data points in our dataset. They typically lead to anomalies in the output of our analysis. Binning can reduce this effect by placing the range of numerical values including the outliers into specific buckets, thereby making the values categorical. A common example of this is when we convert age values into age groups. Outlier ages such as 0 or 150 can fall into a less than age 18 bin and greater than age 80 bin respectively.

In `pandas`, the `cut` method can be used to bin a dataset.

Getting ready

We will work with the full Marketing Campaign data for this recipe.

How to do it...

We will categorize data using the `pandas` library:

1. Import the `pandas` library:

```
import pandas as pd
```

2. Load the `.csv` file into a dataframe using `read_csv`. Then, subset the dataframe to include only relevant columns:

```
marketing_data = pd.read_csv("data/marketing_campaign.
csv")

marketing_data = marketing_data[['ID','Year_
Birth', 'Education','Marital_
Status','Income','Kidhome', 'Teenhome', 'Dt_
Customer',              'Recency','NumStorePurchases',
'NumWebVisitsMonth']]
```

3. Inspect the data. Check the first few rows and use `transpose` (`T`) to show more information. Also, check the data types as well as the number of columns and rows:

```
marketing_data.head(2).T
            0    1
ID    5524    2174
Year_Birth    1957    1954
Education    Graduation    Graduation
…         …         …
NumWebVisitsMonth    7    5

marketing_data.dtypes
ID    int64
Year_Birth    int64
Education    object
…          …
NumWebVisitsMonth    int64

marketing_data.shape
(2240, 11)
```

4. Categorize the number of store purchases into high, moderate, and low categories:

```
marketing_data['bins'] = pd.cut(x=marketing_
data['NumStorePurchases'], bins=[0,4,8,13],labels =
['Low', 'Moderate', 'High'])
```

5. Inspect the result. Subset for relevant columns:

```
marketing_data[['NumStorePurchases','bins']].head()
    NumStorePurchases    bins
0    4    Low
1    2    Low
2    10    High
3    4    Low
4    6    Moderate
```

We have now categorized our dataset into bins.

How it works...

We refer to `pandas` as `pd` in *step 1*. In *step 2*, we use `read_csv` to load the `.csv` file into a `pandas` dataframe and call it `marketing_data`. We also subset the dataframe to include only 11 relevant columns. In *step 3*, we inspect the dataset using `head(2)` to see the first two rows in the dataset; we also use `transpose(T)` along with `head` to transform the rows into columns due to the size of the data (i.e., it has many columns). We use the `dtypes` attribute of the dataframe to show the data types of all columns. Numeric data has int and float data types while character data has the object data type. We inspect the number of rows and columns using `shape`, which returns a tuple that displays the number of rows and columns respectively.

In *step 4*, we categorize the number of store purchases into three categories, namely `High`, `Moderate`, and `Low`. Using the `cut` method, we cut `NumStorePurchases` into these three bins and supply the logic for binning within the `bin` parameter, which is the second parameter. The third parameter is the `label` parameter. Whenever we manually supply the bin edges in a list as done previously, the bins are typically the number of label categories + 1.

Our bins can be interpreted as 0–4 (low), 5–8 (moderate), and 9–13 (high). In *step 5*, we subset for relevant columns and inspect the result of our binning.

There's more...

For the bin argument, we can also supply the number of bins we require instead of supplying the bin edges manually. This means in the previous steps, we could have supplied the value `3` to the `bin` parameter and the `cut` method would have categorized our data into three equally spaced bins. When the value `3` is supplied, the `cut` method focuses on the equal spacing of the bins, even though the number of records in the bins may be different.

If we are also interested in the distribution of our bins and not just equally spaced bins or user-defined bins, the `qcut` method in `pandas` can be used. The `qcut` method ensures the distribution of data in the bins is equal. It ensures all bins have (roughly) the same number of observations, even though the bin range may vary.

Removing duplicate data

Duplicate data can be very misleading and can lead us to wrong conclusions about patterns and the distribution of our data. Therefore, it is very important to address duplicate data within our dataset before embarking on any analysis. Performing a quick duplicate check is good practice in EDA. When working with tabular datasets, we can identify duplicate values in specific columns or duplicate records (across multiple columns). A good understanding of our dataset and the domain will give us insight into what should be considered a duplicate. In `pandas`, the `drop_duplicates` method can help us with handling duplicate values or records within our dataset.

Getting ready

We will work with the full *Marketing Campaign* data for this recipe.

How to do it...

We will remove duplicate data using the `pandas` library:

1. Import the `pandas` library:

```
import pandas as pd
```

2. Load the `.csv` file into a dataframe using `read_csv`. Then, subset the dataframe to include only relevant columns:

```
marketing_data = pd.read_csv("data/marketing_campaign.
csv")
marketing_data = marketing_data[['Education','Marital_
Status','Kidhome', 'Teenhome']]
```

3. Inspect the data. Check the first few rows. Also, check the number of columns and rows:

```
marketing_data.head()
        Education    Marital_Status    Kidhome    Teenhome
0    Graduation    Single    0    0
1    Graduation    Single    1    1
2    Graduation    Together    0    0
3    Graduation    Together    1    0
```

```
4    PhD    Married    1    0

marketing_data.shape
(2240, 4)
```

4. Remove duplicates across the four columns in our dataset:

```
marketing_data_duplicate = marketing_data.drop_
duplicates()
```

5. Inspect the result:

```
marketing_data_duplicate.head()
    Education    Marital_Status    Kidhome    Teenhome
0    Graduation    Single    0    0
1    Graduation    Single    1    1
2    Graduation    Together    0    0
3    Graduation    Together    1    0
4    PhD    Married    1    0

marketing_data_duplicate.shape
(135,4)
```

We have now removed duplicates from our dataset.

How it works...

We refer to `pandas` as `pd` in *step 1*. In *step 2*, we use `read_csv` to load the `.csv` file into a `pandas` dataframe and call it `marketing_data`. We also subset the dataframe to include only four relevant columns. In *step 3*, we inspect the dataset using `head()` to see the first five rows in the dataset. Using the `shape` method, we get a sense of the number of rows and columns from the tuple respectively.

In *step 4*, we use the `drop_duplicates` method to remove duplicate rows that appear in the four columns of our dataset. We save the result in the `marketing_data_duplicate` variable. In *step 5*, we inspect the result using the `head` method to see the first five rows. We also leverage the `shape` method to inspect the number of rows and columns. We can see that the rows have decreased significantly from our original shape.

There's more...

The `drop_duplicates` method gives some flexibility around dropping duplicates based on a subset of columns. By supplying the list of the subset columns as the first argument, we can drop all rows that contain duplicates based on those subset columns. This is useful when we have several columns and only a few key columns contain duplicate information. Also, it allows us to keep instances of duplicates, using the `keep` parameter. With the `keep` parameter, we can specify whether we want to keep the "first" or "last" instance or drop all instances of the duplicate information. By default, the method keeps the first instance.

Dropping data rows and columns

When working with tabular data, we may have reason to drop some rows or columns within our dataset. Sometimes, we may need to drop columns or rows either because they are erroneous or irrelevant. In `pandas`, we have the flexibility to drop a single row/column or multiple rows/columns. We can use the `drop` method to achieve this.

Getting ready

We will work with the full *Marketing Campaign* data for this recipe.

How to do it...

We will drop rows and columns using the `pandas` library:

1. Import the `pandas` library:

```
import pandas as pd
```

2. Load the `.csv` file into a dataframe using `read_csv`. Then, subset the dataframe to include only relevant columns:

```
marketing_data = pd.read_csv("data/marketing_campaign.
csv")
marketing_data = marketing_data[['ID', 'Year_Birth',
'Kidhome', 'Teenhome']]
```

3. Inspect the data. Check the first few rows. Check the number of columns and rows:

```
marketing_data.head()
    ID    Year_Birth    Education    Marital_Status
0    5524    1957    Graduation    Single
1    2174    1954    Graduation    Single
```

```
2    4141    1965    Graduation    Together
3    6182    1984    Graduation    Together
4    5324    1981    PhD    Married

marketing_data.shape
(5, 4)
```

4. Delete a specified row at index value `1`:

```
marketing_data.drop(labels=[1], axis=0)
    ID    Year_Birth    Education    Marital_Status
0    5524    1957    Graduation    Single
2    4141    1965    Graduation    Together
3    6182    1984    Graduation    Together
4    5324    1981    PhD    Married
```

5. Delete a single column:

```
marketing_data.drop(labels=['Year_Birth'], axis=1)
    ID    Education    Marital_Status
0    5524    Graduation    Single
1    2174    Graduation    Single
2    4141    Graduation    Together
3    6182    Graduation    Together
4    5324    PhD    Married
```

Good job! We have dropped rows and columns from our dataset.

How it works...

We refer to `pandas` as `pd` in *step 1*. In *step 2*, we use `read_csv` to load the `.csv` file into a `pandas` dataframe and call it `marketing_data`. We also subset the dataframe to include only four relevant columns. In *step 3*, we inspect the dataset using `head()` to see the first five rows in the dataset. Using the `shape` method, we get a sense of the number of rows and columns from the tuple respectively.

In *step 4*, we use the `drop` method to delete a specified row at index value `1` and view the result, which shows the row at index `1` has been removed. The `drop` method takes a list of indices as the first argument and an `axis` value as the second. The `axis` value determines whether the drop operation will be performed on a row or column. A value of `0` is used for rows while `1` is used for columns.

In *step 5*, we use the `drop` method to delete a specified column and view the result, which shows the specific column has been removed. To drop columns, we need to specify the name of the column and provide the `axis` value of `1`.

There's more...

We can drop multiple rows or columns using the `drop` method. To achieve this, we need to specify all the row indices or column names in a list and provide the respective axis value of `0` or `1` for rows and columns respectively.

Replacing data

Replacing values in rows or columns is a common practice when working with tabular data. There are many reasons why we may need to replace specific values within a dataset. Python provides the flexibility to replace single values or multiple values within our dataset. We can use the `replace` method to achieve this.

Getting ready

We will work with the *Marketing Campaign* data again for this recipe.

How to do it...

We will remove duplicate data using the `pandas` library:

1. Import the `pandas` library:

```
import pandas as pd
```

2. Load the `.csv` file into a dataframe using `read_csv`. Then, subset the dataframe to include only relevant columns:

```
marketing_data = pd.read_csv("data/marketing_campaign.
csv")
marketing_data = marketing_data[['ID', 'Year_Birth',
'Kidhome', 'Teenhome']]
```

3. Inspect the data. Check the first few rows, and check the number of columns and rows:

```
     ID    Year_Birth    Kidhome    Teenhome
0    5524    1957    0    0
1    2174    1954    1    1
2    4141    1965    0    0
```

```
3    6182    1984    1    0
4    5324    1981    1    0

marketing_data.shape
(2240, 4)
```

4. Replace the values in `Teenhome` with `has teen` and `has no teen`:

```
marketing_data['Teenhome_replaced'] = marketing_
data['Teenhome'].replace([0,1,2],['has no teen','has
teen','has teen'])
```

5. Inspect the output:

```
marketing_data[['Teenhome','Teenhome_replaced']].head()
    Teenhome    Teenhome_replaced
0    0    has no teen
1    1    has teen
2    0    has no teen
3    0    has no teen
4    0    has no teen
```

Great! We just replaced values in our dataset.

How it works...

We refer to `pandas` as *pd* in *step 1*. In *step 2*, we use `read_csv` to load the `.csv` file into a `pandas` dataframe and call it `marketing_data`. We also subset the dataframe to include only four relevant columns. In *step 3*, we inspect the dataset using `head()` to see the first five rows in the dataset. Using the `shape` method, we get a sense of the number of rows and columns.

In *step 4*, we use the `replace` method to replace values within the `Teenhome` column. The first argument of the method is a list of the existing values that we want to replace, while the second argument contains a list of the values we want to replace it with. It is important to note that the lists for both arguments must be the same length.

In *step 5*, we inspect the result.

There's more...

In some cases, we may need to replace a group of values that have complex patterns that cannot be explicitly stated. An example could be certain phone numbers or email addresses. In such cases, the `replace` method gives us the ability to use regex for pattern matching and replacement. **Regex** is short for **regular expressions**, and it is used for pattern matching.

See also

- You can check out this great resource by *Data to Fish* on replacing data in `pandas`: `https://datatofish.com/replace-values-pandas-dataframe/`
- Here is an insightful resource by *GeeksforGeeks* on regex in the `replace` method in `pandas`: `https://www.geeksforgeeks.org/replace-values-in-pandas-dataframe-using-regex/`
- Here is another article by *W3Schools* that highlights common regex patterns in Python: `https://www.w3schools.com/python/python_regex.asp`

Changing a data format

When analyzing or exploring data, the type of analysis we perform on our data is highly dependent on the data formats or data types within our dataset. Typically, numerical data requires specific analytical techniques, while categorical data requires other analytical techniques. Hence, it is important that data types are properly captured before analysis commences.

In `pandas`, the `dtypes` attribute helps us to inspect the data types within our dataset, while the `astype` attribute helps us to convert our dataset between various data types.

Getting ready

We will work with the *Marketing Campaign* data again for this recipe.

How to do it...

We will change the format of our data using the `pandas` library:

1. Import the `pandas` library:

```
import pandas as pd
```

2. Load the `.csv` file into a dataframe using `read_csv`. Then, subset the dataframe to include only relevant columns:

```
marketing_data = pd.read_csv("data/marketing_campaign.
csv")
marketing_data = marketing_data[['ID', 'Year_
Birth','Marital_Status','Income']]
```

3. Inspect the data. Check the first few rows. Check the number of columns and rows:

```
    ID    Year_Birth    Marital_
Status    Income    Income_changed
0    5524    1957    Single    58138.0    58138
1    2174    1954    Single    46344.0    46344
2    4141    1965    Together    71613.0    71613
3    6182    1984    Together    26646.0    26646
4    5324    1981    Married    58293.0    58293

marketing_data.shape
(2240, 5)
```

4. Fill NAs in the `Income` column:

```
marketing_data['Income'] = marketing_data['Income'].
fillna(0)
```

5. Change the data type of the `Income` column from float to int:

```
marketing_data['Income_changed'] = marketing_
data['Income'].astype(int)
```

6. Inspect the output using the `head` method and `dtypes` attribute:

```
marketing_data[['Income','Income_changed']].head()
    Income    Income_changed
0    58138.0    58138
1    46344.0    46344
2    71613.0    71613
3    26646.0    26646
4    58293.0    58293

marketing_data[['Income','Income_changed']].dtypes
```

```
                0
Income          float64
Income_changed    int32
```

Now we have changed the format of our dataset.

How it works...

We refer to `pandas` as `pd` in *step 1*. In *step 2*, we use `read_csv` to load the csv file into a `pandas` dataframe and call it `marketing_data`. We also subset the dataframe to include only four relevant columns. In *step 3*, we inspect the dataset using `head()` to see the first five rows in the dataset. Using the `shape` method, we get a sense of the number of rows and columns.

In *step 4*, we fill NaN values with zeros using the `fillna` method. This is an important step before we can change data types in `pandas`. We have provided the `fillna` method with the `just` argument, which is the value to replace NaN values with. In *step 5*, we change the data type of the `Income` column from float to int using the `astype` method. We supply the data type we wish to convert to as the argument for the method.

In *step 6*, we subset the dataframe and inspect the result.

There's more...

When converting data types, we may encounter errors in conversion. The `astype` method gives us options through the `errors` parameter to raise or ignore errors. By default, the method raises errors; however, we can ignore errors so that the method returns the original values for each error identified.

See also

Here is a great article by *PB Python* that provides more details on converting data types in `pandas`: `https://pbpython.com/pandas_dtypes.html`.

Dealing with missing values

Dealing with missing values is a common problem we will typically face when analyzing data. A missing value is a value within a field or variable that is not present, even though it is expected to be. There are several reasons why this could have happened, but a common reason is that the data value wasn't provided at the point of data collection. As we explore and analyze data, missing values can easily lead to inaccurate or biased conclusions; therefore, they need to be taken care of. Missing values are typically represented by blank spaces, but in `pandas`, they are represented by NaN.

Several techniques can be used to deal with missing values. In this recipe, we will focus on dropping missing values using the `dropna` method in `pandas`.

Getting ready

We will work with the full *Marketing Campaign* data in this recipe.

How to do it...

We will drop rows and columns using the `pandas` library:

1. Import the `pandas` library:

```
import pandas as pd
```

2. Load the `.csv` into a dataframe using `read_csv`. Then, subset the dataframe to include only relevant columns:

```
marketing_data = pd.read_csv("data/marketing_campaign.
csv")
marketing_data = marketing_data[['ID', 'Year_Birth',
'Education','Income']]
```

3. Inspect the data. Check the first few rows, and check the number of columns and rows:

```
marketing_data.head()
    ID    Year_Birth        Education    Income
0    5524    1957            Graduation    58138.0
1    2174    1954            Graduation    46344.0
2    4141    1965            Graduation    71613.0
3    6182    1984            Graduation    26646.0
4    5324    1981            PhD         58293.0

marketing_data.shape
(2240, 4)
```

4. Check for missing values using the `isnull` and `sum` methods:

```
marketing_data.isnull().sum()
ID              0
Year_Birth      0
Education       0
Income         24
```

5. Drop missing values using the `dropna` method:

```
marketing_data_withoutna = marketing_data.dropna(how =
'any')
marketing_data_withoutna.shape
(2216, 4)
```

Good job! We have dropped missing values from our dataset.

How it works...

In *step 1*, we import `pandas` and refer to it as `pd`. In *step 2*, we use `read_csv` to load the `.csv` file into a `pandas` dataframe and call it `marketing_data`. We also subset the dataframe to include only four relevant columns. In *step 3*, we inspect the dataset using the `head` and `shape` methods.

In *step 4*, we use the `isnull` and `sum` methods to check for missing values. These methods give us the number of rows with missing values within each column in our dataset. Columns with zero have no rows with missing values.

In *step 5*, we use the `dropna` method to drop missing values. For the `how` parameter, we supply `'any'` as the value to indicate that we want to drop rows that have missing values in any of the columns. An alternative value to use is `'all'`, which ensures all the columns of a row have missing values before dropping the row. We then check the number of rows and columns using the `shape` method. We can see that the final dataset has 24 fewer rows.

There's more...

As highlighted previously, there are several reasons why a value may be missing in our dataset. Understanding the reason can point us to optimal solutions to resolve this problem. Missing values shouldn't be addressed with a one-size-fits-all approach. *Chapter 9* dives into the details of how to optimally deal with missing values and outliers by providing several techniques.

See also

You can check out a detailed approach to dealing with missing values and outliers in *Chapter 9*, *Dealing with Outliers and Missing Values*.