Module 4 Part 2: Introduction to Pandas

This module consists of 3 parts:

- **Part 1** - Object-Oriented Programming with Python and Additional Python Functions
- **Part 2** - Introduction to pandas
- **Part 3** - Modifying a DataFrame, data aggregation and grouping, Case Studies

Each part is provided in a separate notebook file. It is recommended that you follow the order of the notebooks.

Readings and Resources

The majority of the notebook content draws from the recommended readings. We invite you to further supplement this notebook with the following recommended texts:

McKinney, W. (2017). *Python for Data Analysis*. O-Reilly: Boston

Table of Contents

- EXERCISE 4: Loading data into DataFrame, exploring
- References

Pandas

Prepare the Data: Pandas

> This image shows the stages of creating a model from start to finish. (Course Authors, 2018)
> *This image shows the stages of creating a model from start to finish.*

In the last module, you learned how to create matrices and arrays using the NumPy library and received a brief introduction to applying basic mathematical functions to your data. In this module, we will learn about the Pandas library available in Python.

In this phase of the analytics methodology — Preparing the Data — we want to ensure the data is not only organized appropriately, but also labelled in a logical manner so we can understand what data points constitute our sample or population, so that we can conduct preliminary analyses on our data. The Pandas library allows us to organize our data in a Data Frame. You can think of a Data Frame as a chart with column and row headings. Structuring our data this way will enable us to understand basic characteristics of big data sets. For example, we can easily identify how many empty values there are, or how the data is segmented.

This is also referred to as data exploration. Prior to creating predictive or prescriptive models, it is critical to explore your data and ensure there is no obvious bias, or significant missing data points that may skew your results. After all, you will be training your model based on a portion of your data set, so it is critical to have data that is both representative of your population and as complete and accurate as possible.

Pandas is a data analysis package created by Wes McKinney. It brings an equivalent of the R Data Frame to Python. Pandas is a powerful package that defines data structures and tools for easy and intuitive data analysis.

Pandas gets its name from the term "panel data" which used to refer to multi-dimensional structured data sets.

Pandas was built on NumPy, which, as we know from the previous module stores data in arrays. The main difference is that NumPy mostly works with homogeneous numerical arrays while pandas works with heterogeneous or tabular data.

To start working with the `pandas` package, we will need to import it as follows:

In [1]:
```
import pandas as pd
```

Series and **DataFrame** are two major data structures that we will explain in this module.

Series

A pandas **Series** is a one-dimensional array that can hold indexed data of any type *(integers, strings, floating point numbers, Python objects, etc.)*. Series can be created using:

- Python dictionaries
- NumPy ndarrays
- Scalar values
- Lists

Let's start by creating a pandas Series using a list:

In [2]:
```
list_ser = [45, 123, 67, 1, 14]
```

In [3]:
```
serA = pd.Series(list_ser)
serA
```

Out[3]:
```
0     45
1    123
2     67
3      1
4     14
dtype: int64
```

In [4]:
```
type(serA)
```

Out[4]:
```
pandas.core.series.Series
```

Pandas created a `Series` using the list `list_ser`, the `dtype` of the `Series` is `int64`. The type of data was inferred since we didn't provide a parameter and didn't specify explicitly what type of data we are planning to use. `pandas` determined that all numbers in the original list were integers, making the resulting `Series` a 64-bit integer.

The first column of numbers from 0 to 4 is the **index**. When we created a `Series`, we didn't specify if we want an index to be something specific, hence `pandas` used default values.

NOTE: The default index in pandas **always starts with 0 (zero)**.

We can inspect an index by using the following command:

In [5]:
```
serA.index
```

Out[5]:
```
RangeIndex(start=0, stop=5, step=1)
```

In this example, the index is a range of integers from 0 to 5 (excluded) with a step of 1.

While creating a `Series`, in addition to the data source, we can specify the index, set the name of the `Series` and explicitly define the type of data:

```
pandas.Series(data=None, index=None, dtype=None, name=None,
copy=False, fastpath=False)
```

The `Series` documentation is a handy reference. You may want to bookmark it for future reference.

In the next example, we will create a `Series` using the same list as above, but we will define an index, data type and `Series` name:

In [6]:
```
serB = pd.Series(list_ser,
                 index=['Num1','Num2','Num3','Num4','Num5'],  # indices can be stri
                 dtype='float',      # even though the list contains integers, we wa
                 name='Numbers')
serB
```

Out[6]:
```
Num1     45.0
Num2    123.0
Num3     67.0
Num4      1.0
Num5     14.0
Name: Numbers, dtype: float64
```

Each data point in `serB` has a label associated with it:

In [7]:
```
serB.index
```

Out[7]:
```
Index(['Num1', 'Num2', 'Num3', 'Num4', 'Num5'], dtype='object')
```

In [8]:
```
serB.values
```

Out[8]:
```
array([ 45., 123.,  67.,   1.,  14.])
```

In the next example we will create a `Series` from a dictionary. We will use the top 5 Canadian provinces by population (retrieved from Statistics Canada web site, we used the 2017 column of data):

In [9]:
```
population_dict = {'ON': 14193384, 'QC': 8394034, 'BC': 4817160, 'AB': 4286134, 'MB

provinces_population = pd.Series(population_dict, name='Top 5 provinces by populati
provinces_population
```

Out[9]:
```
ON    14193384
QC     8394034
BC     4817160
AB     4286134
MB     1338109
Name: Top 5 provinces by population, dtype: int64
```

When a dictionary is used to create a `Series`, dictionary keys become indices.

The index of a `Series` can be represented by dates. In this case, we are talking about time series. We will talk about time series later in the course.

We can use the index to select values from a `Series`. Here are several examples:

In [10]:
```
# Population of Ontario
provinces_population['ON']
```

Out[10]:
```
14193384
```

In [11]:
```
# Selecting only provinces with population greater than 5 million.
# This type of selection is called boolean indexing:

provinces_population[provinces_population > 5000000]
```

Out[11]:
```
ON    14193384
QC     8394034
Name: Top 5 provinces by population, dtype: int64
```

In [12]:
```
# Entries with index positions of 2 and 3, should return British Columbia and Alber
provinces_population[2:4]
```

Out[12]:
```
BC    4817160
AB    4286134
Name: Top 5 provinces by population, dtype: int64
```

As you can see, even though the indices are strings, `pandas` allows us to use the integer position of an index key. Please remember that integer position starts with zero.

If we use the index keys for a slice operation, both parameters are inclusive and the value for Manitoba is returned.

In [13]:
```
# The result of this command should be 3 provinces,
# British Columbia, Alberta and Manitoba

provinces_population['BC':'MB']
```

Out[13]:
```
BC    4817160
AB    4286134
MB    1338109
Name: Top 5 provinces by population, dtype: int64
```

We can check if the `Series` has specific values we are looking for. For example, we might check if Quebec is within the top 5 provinces (by population):

In [14]:
```
'QC' in provinces_population
```

Out[14]:
```
True
```

In [15]:
```
# And what about Nova Scotia:

'NS' in provinces_population
```

Out[15]:
```
False
```

Pandas series has several built-in functions that allow us to do mathematical operations on the elements within the series. For example, we can summarize all the elements within a series with the `sum()` function, or calculate the mean with `mean()` function:

In [16]:
```
provinces_population.sum()
```

Out[16]:
```
33028821
```

In [17]:
```
provinces_population.mean()
```

Out[17]:
```
6605764.2
```

EXERCISE 3: Series manipulation

Task 1: Create a new `Series` object that will contain the population of Nova Scotia and New Brunswick. Use the data from the Statistics Canada website.

In [20]:
```
list_add = [953869,759655]

pop_add = pd.Series(list_add, index = ['NS', 'NB'],name='Nova Scotia and New Brunsw

pop_add

print(pd.__version__)
```

```
2.1.1
```

Task 2: Merge your new `Series` and the `provinces_population` into a single `Series` object. Use the `append()` function

In [21]:
```
provinces_population = pd.concat([provinces_population, pop_add])
```

In [22]:
```
provinces_population
```

Out[22]:
```
ON    14193384
QC     8394034
BC     4817160
AB     4286134
MB     1338109
NS      953869
NB      759655
dtype: int64
```

Solution - Task 1. We can create a new `Series` object as we did above, using a dictionary:

In [24]:
```
NS_NB_dict = {'NS': 953869, 'NB': 759655}

NS_NB_population = pd.Series(NS_NB_dict)
NS_NB_population
```

Out[24]:
```
NS    953869
NB    759655
dtype: int64
```

Solution - Task 2. Appending `NS_NB_population` Series to `provinces_population`:

In [25]:
```
provinces_population = pd.concat([provinces_population,NS_NB_population])
provinces_population
```

Out[25]:
```
ON    14193384
QC     8394034
BC     4817160
AB     4286134
MB     1338109
NS      953869
NB      759655
NS      953869
NB      759655
dtype: int64
```

DataFrames

A pandas **DataFrame** is a *2-dimensional tabular data structure with labeled columns and rows*. Columns in a `DataFrame` can be of different data types.

You can think of a `DataFrame` as a group of `pandas` `Series` where each `Series` represents a column of data. You can also think of a `DataFrame` as a collection of columns with the same index rather than a collection of rows. This view will also help in understanding some of the `pandas` `DataFrame` functionality.

Creating a DataFrame

A `DataFrame` can be created from a:

- Dictionary of 1-D structures (`ndarray` s, `list` s, dictionaries, tuples or `Series`)

- List of 1-D structures
- 2-D NumPy `ndarray`
- `Series`
- Another `DataFrame`

As an example, here is how we would create a `DataFrame` from a 2-dimensional list:

In [26]:
```
df = pd.DataFrame(data=[[8, 128, 27.5],
                        [10, 138.9, 34.5],
                        [16, 157.3, 91.1],
                        [6, 116.6, 21.4],
                        [14, 159.2, 54.4]],
                  columns=['Age', 'Height', 'Weight'])
```

In [27]:
```
df
```

Out[27]:

	Age	Height	Weight
0	8	128.0	27.5
1	10	138.9	34.5
2	16	157.3	91.1
3	6	116.6	21.4
4	14	159.2	54.4

Since we didn't provide any index values for rows, `pandas` used default indices, from 0 to 4. The column labels were defined by the `columns` parameter.

We can inspect the `DataFrame` and use the `info()` function to output summary information about the `DataFrame`:

In [23]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 3 columns):
Age       5 non-null int64
Height    5 non-null float64
Weight    5 non-null float64
dtypes: float64(2), int64(1)
memory usage: 200.0 bytes
```

This method allows us to quickly examine the `DataFrame` and determine the following:

- There are 5 rows of data in the `DataFrame` with the index as a range of integers from 0 to 4
- The `DataFrame` has 3 columns of data, column names are `Age`, `Height` and `Weight`

- All columns have 5 records and there are no Null values in any of the columns
- The first column, `Age`, is a column of integers, the second and third columns, `Height` and `Weight`, are floating point numbers, which is also stated in the `dtypes: float64(2), int64(1)` line
- The last line will show approximately how many bytes of memory are used by `pandas` to store this `DataFrame`

We can also use the following methods to further examine the `DataFrame`:

In [24]:
```
# Index labels
df.index
```

Out[24]:
```
RangeIndex(start=0, stop=5, step=1)
```

In [25]:
```
# Columns labels
df.columns
```

Out[25]:
```
Index(['Age', 'Height', 'Weight'], dtype='object')
```

To create a `DataFrame` from a dictionary, let's start with a simple dictionary which will contain area data for land and water areas for the five Canadian provinces that we worked with in the previous example (i.e. Ontario, Quebec, British Columbia, Alberta and Manitoba). We will retrieve our data from a Wikipedia page.

In [26]:
```
area = {'province':['ON', 'QC', 'BC', 'AB', 'MB'],
        'area_land': [917741, 1356128, 925186, 642317, 553556],
        'area_water': [158654, 185928, 19549, 19531, 94241]}
```

The easiest way to create a `DataFrame` is to use the `area` dictionary as the only parameter:

In [27]:
```
provinces_area = pd.DataFrame(area)
provinces_area
```

Out[27]:

	province	area_land	area_water
0	ON	917741	158654
1	QC	1356128	185928
2	BC	925186	19549
3	AB	642317	19531
4	MB	553556	94241

As you can see, `pandas` created a new `DataFrame` with 3 columns and the index set to the default range of integers from 0 to 4. If we want the `province` to be an index for this `DataFrame`, we need to use the method `set_index()`:

In [28]:
```
provinces_area = provinces_area.set_index('province')
provinces_area
```

Out[28]:

province	area_land	area_water
ON	917741	158654
QC	1356128	185928
BC	925186	19549
AB	642317	19531
MB	553556	94241

In the next section, we will cover reading data from a file into a `DataFrame` and writing data from a `DataFrame` to a file.

Loading/Saving DataFrames

During your career as a data scientist, you will be analysing data sourced from different systems and stored in many different formats. Some examples of data formats are:

- `csv` format or as plain text (`.txt` or no extension plain text)
- Excel files (i.e., `.xlsx`)
- Documents in `.doc` or `.pdf` formats
- Data retrieved from relational and noSQL datastores

Most of the time, regardless of the source of the data and its current format, one of the first steps in data analysis is to load the data into a `pandas` `DataFrame` and perform exploratory analysis.

In this section, we will learn how to read the data stored in `.csv` format and load it into a `DataFrame` . We will also learn how to save a `DataFrame` to `.csv` format after the processing of the data is complete.

Loading DataFrames

For this exercise, we will continue looking into the data that describes Canadian provinces. This time, we will use the data of the last 3 years of Federal Support to all Canadian Provinces and Territories. All numbers are in millions of dollars.

Pandas has a `read_csv()` function which we will use. There are quite a few parameters that can be specified to either filter out the characters or even lines within the file which are not relevant or make sure that the data is read in a proper format. For example, by default, the `read_csv()` function expects the column separator to be a comma, but you can change that using the `sep=` parameter, or you can skip rows with the `skiprows=` parameter. A complete list of all parameters can be found on the corresponding documentation page for the `read_csv()` function.

The dataset that we are planning to use in this section is stored in the `pandas_ex1.csv` file, we assume that the file is in the same folder as the notebook working directory. You can open the file in any text editor and validate the data. The first six rows will look as follows:

```
Canadian Provinces and Territories,Two-Letter Abbreviation,2016-
17,2017-18,2018-19
Newfoundland and Labrador,NL,724,734,750
Prince Edward Island,PE,584,601,638
Nova Scotia,NS,3060,3138,3201
New Brunswick,NB,2741,2814,2956
Quebec,QC,21372,22720,23749
```

As you can see, the data is formatted perfectly, it contains what looks like a header row and the fields are separated by commas. In this case, we can simply specify the name of the file, and include only one more parameter, `sep=`, to ensure that the Python interpreter reads the data correctly:

In [28]:
```
# Reading a csv
prov_support = pd.read_csv('pandas_ex1.csv', sep=',')
prov_support
```

Out[28]:

	Canadian Provinces and Territories	Two-Letter Abbreviation	2016-17	2017-18	2018-19
0	Newfoundland and Labrador	NL	724	734	750
1	Prince Edward Island	PE	584	601	638
2	Nova Scotia	NS	3060	3138	3201
3	New Brunswick	NB	2741	2814	2956
4	Quebec	QC	21372	22720	23749
5	Ontario	ON	21347	21101	21420
6	Manitoba	MB	3531	3675	3965
7	Saskatchewan	SK	1565	1613	1673
8	Alberta	AB	5772	5943	6157
9	British Columbia	BC	6482	6680	6925
10	Yukon	YT	946	973	1006
11	Northwest Territories	NT	1281	1294	1319
12	Nunavut	NU	1539	1583	1634

A couple of observations:

1. The first row in the file was indeed a header row
2. Pandas created a `DataFrame` with an index from 0 to 12

Let's check the summary information of the `DataFrame` object:

In [30]:
```
prov_support.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 13 entries, 0 to 12
Data columns (total 5 columns):
Canadian Provinces and Territories    13 non-null object
Two-Letter Abbreviation               13 non-null object
2016-17                               13 non-null int64
2017-18                               13 non-null int64
2018-19                               13 non-null int64
dtypes: int64(3), object(2)
memory usage: 600.0+ bytes
```

We can see that the `DataFrame` object has 5 columns of data, 3 columns are integers and 2 columns are of `object` data type. In pandas, "object" usually means `string`. We can also use the method `dtypes` to check data types for each column of data:

In [31]:
```
prov_support.dtypes
```

Out[31]:
```
Canadian Provinces and Territories    object
Two-Letter Abbreviation               object
2016-17                                int64
2017-18                                int64
2018-19                                int64
dtype: object
```

When creating a `DataFrame` , `pandas` attempts to infer type and automatically do type conversion on read. This includes user-defined value conversions including custom values for missing value markers.

Type inference is one of the most important features of `pandas` . It means *you do not have to specify which columns are numeric, integer, Boolean, or string*. Pandas will inspect and specify for you the types. However, this can lead to problems if you are not following best practices.

For example, suppose a data set uses numbers to represent nominal category levels instead of string labels. When trying to do an analysis, most statistical packages would default to treating the levels as a numeric, unintentionally implying an ordering and quantity. If these Python analysis packages also make inferences from the data, then it is also possible that the package would default to another methodology. Simple time saving features can quickly spiral into more problems.

Therefore, it is a good practice to always check if `pandas` has read the data correctly. One common problem is when the numeric fields in the data file to be read have leading or trailing spaces, or if the numeric fields contain special characters, for example dollar signs, or thousands separators. Pandas will read such numeric fields as strings. You might run into problems with this data if you try to do any calculations on these columns, or use them in visualizations or models.

We can set more parameters while reading the file. In the previous section, while creating the `DataFrame` from the dictionary, we decided to use one of the columns as an index. We can do the same here and use the column with province abbreviations as an index for the `DataFrame` . Of course, we can always update the `DataFrame` afterwards using the `set_index()` function, but `read_csv()` has a parameter that accomplishes this when reading the data.

We might also decide not to use the header row from the file as it is too verbose, and specify different names for the columns. We will demonstrate below how to add column names, because you might not always have a header in the data file, and you will need to specify a header yourself. It is possible to add it later, but you can also name the columns within `read_csv()` . Let's see how:

In [29]:
```
prov_support = pd.read_csv('pandas_ex1.csv',
                           sep=',',
                           skiprows=1,   # skipping one row
                           header=None,  # we are telling pandas that there is no h
```

```
                                  names=['province_name','province','2016','2017','2018'],
                                  index_col='province')   # use column 'province' as index

prov_support
```

Out[29]:

province	province_name	2016	2017	2018
NL	Newfoundland and Labrador	724	734	750
PE	Prince Edward Island	584	601	638
NS	Nova Scotia	3060	3138	3201
NB	New Brunswick	2741	2814	2956
QC	Quebec	21372	22720	23749
ON	Ontario	21347	21101	21420
MB	Manitoba	3531	3675	3965
SK	Saskatchewan	1565	1613	1673
AB	Alberta	5772	5943	6157
BC	British Columbia	6482	6680	6925
YT	Yukon	946	973	1006
NT	Northwest Territories	1281	1294	1319
NU	Nunavut	1539	1583	1634

We now know how to set both row and column indexes and load the data from a `csv` file into the `DataFrame`.

Saving DataFrames

We now understand how to read data. In this section we will discuss how to write and save data once we've finished preparing data for analysis.

In `pandas`, we utilize the `to_csv()` function for writing out `DataFrame` data in `csv` format. It is similar to the reading function, except now the parameters are used for formatting output instead of parsing input.

In [30]:
```
csv_out = prov_support.to_csv()
csv_out
```

Out[30]:
```
'province,province_name,2016,2017,2018\r\nNL,Newfoundland and Labrador,724,734,750
\r\nPE,Prince Edward Island,584,601,638\r\nNS,Nova Scotia,3060,3138,3201\r\nNB,New
Brunswick,2741,2814,2956\r\nQC,Quebec,21372,22720,23749\r\nON,Ontario,21347,21101,
21420\r\nMB,Manitoba,3531,3675,3965\r\nSK,Saskatchewan,1565,1613,1673\r\nAB,Albert
a,5772,5943,6157\r\nBC,British Columbia,6482,6680,6925\r\nYT,Yukon,946,973,1006\r
\nNT,Northwest Territories,1281,1294,1319\r\nNU,Nunavut,1539,1583,1634\r\n'
```

In [31]:
```python
# writing to a file
prov_support.to_csv('csv_out1.csv')
```

We can also make a more elaborate call and specify if we want to include a header row and index, and what character should be used as a column separator:

In [33]:
```python
prov_support.to_csv('csv_out2.csv',
                    sep='\t',        # the separator used for the columns
                    index=True,      # whether to include indexes in output
                    header=True)     # whether to include headers in output
```

We can find the files on the file system, open them in any text editor and validate the data. The data in the first file, `csv_out1.csv` should be formatted as follows:

```
province,province_name,2016,2017,2018
NL,Newfoundland and Labrador,724,734,750
PE,Prince Edward Island,584,601,638
NS,Nova Scotia,3060,3138,3201
NB,New Brunswick,2741,2814,2956
```

The format of the second file, `csv_out2.csv`, will look as follows — the columns are separated by tabs instead of commas:

```
province        province_name   2016    2017    2018
NL      Newfoundland and Labrador       724     734     750
PE      Prince Edward Island    584     601     638
NS      Nova Scotia     3060    3138    3201
NB      New Brunswick   2741    2814    2956
```

Renaming DataFrame columns

Quite often, after you have read the file and created a `DataFrame`, you realize that the column names are either too long, contain special characters, or are not descriptive enough.

In our example, we might decide to shorten the names of the first and second columns. We might want to rename "Canadian Provinces and Territories" to "Province Name" and "Two-Letter Abbreviation" to "Province Abbreviation". We also might want to rename the numeric columns to display only one year instead of the range of years. For example, rename `"2016-17"` to be `"2016"` and so on.

Let's load the data into the `DataFrame` one more time without specifying any parameters for the `read_csv()` function, like we did in the beginning of this section:

In [34]:
```
fed_sup = pd.read_csv('pandas_ex1.csv', sep=',')
fed_sup
```

Out[34]:

	Canadian Provinces and Territories	Two-Letter Abbreviation	2016-17	2017-18	2018-19
0	Newfoundland and Labrador	NL	724	734	750
1	Prince Edward Island	PE	584	601	638
2	Nova Scotia	NS	3060	3138	3201
3	New Brunswick	NB	2741	2814	2956
4	Quebec	QC	21372	22720	23749
5	Ontario	ON	21347	21101	21420
6	Manitoba	MB	3531	3675	3965
7	Saskatchewan	SK	1565	1613	1673
8	Alberta	AB	5772	5943	6157
9	British Columbia	BC	6482	6680	6925
10	Yukon	YT	946	973	1006
11	Northwest Territories	NT	1281	1294	1319
12	Nunavut	NU	1539	1583	1634

Pandas allows us to easily rename the columns in one line of code. All we need to do is to use the `.columns` attribute and pass to it a list of new column names:

In [37]:
```
fed_sup.columns = ['Province Name', 'Province Abbreviation', '2016', '2017', '2018'
fed_sup
```

Out[37]:

	Province Name	Province Abbreviation	2016	2017	2018
0	Newfoundland and Labrador	NL	724	734	750
1	Prince Edward Island	PE	584	601	638
2	Nova Scotia	NS	3060	3138	3201
3	New Brunswick	NB	2741	2814	2956
4	Quebec	QC	21372	22720	23749
5	Ontario	ON	21347	21101	21420
6	Manitoba	MB	3531	3675	3965
7	Saskatchewan	SK	1565	1613	1673
8	Alberta	AB	5772	5943	6157
9	British Columbia	BC	6482	6680	6925
10	Yukon	YT	946	973	1006
11	Northwest Territories	NT	1281	1294	1319
12	Nunavut	NU	1539	1583	1634

Now, what if we decide that we don't want to have spaces in the column names (we will discuss in the next section the advantages of having column names without spaces). In this case, we can use `pandas`' `rename()` function:

In [38]:
```
fed_sup = fed_sup.rename(columns={'Province Name':'province_name', 'Province Abbrev
fed_sup
```

Out[38]:

	province_name	province	2016	2017	2018
0	Newfoundland and Labrador	NL	724	734	750
1	Prince Edward Island	PE	584	601	638
2	Nova Scotia	NS	3060	3138	3201
3	New Brunswick	NB	2741	2814	2956
4	Quebec	QC	21372	22720	23749
5	Ontario	ON	21347	21101	21420
6	Manitoba	MB	3531	3675	3965
7	Saskatchewan	SK	1565	1613	1673
8	Alberta	AB	5772	5943	6157
9	British Columbia	BC	6482	6680	6925
10	Yukon	YT	946	973	1006
11	Northwest Territories	NT	1281	1294	1319
12	Nunavut	NU	1539	1583	1634

The advantage of using the `rename()` function is that you can include only those column names that require the change and be specific about what column names you want to rename.

Working with DataFrames

In this section, we will learn how to access, select and manipulate data in `pandas` `DataFrame` s.

We have read the data from the file and loaded it into a `DataFrame` . If you have thousands or even millions of rows of data, you might want to look at a small sample of the data within the notebook. Pandas has two very convenient methods to do that, `head()` and `tail()` . Both functions, by default, will display only 5 rows of data, the first or the last 5 rows, correspondingly.

In [39]:
```
# first 5 rows of the DataFrame
prov_support.head()
```

Out[39]:

	province_name	2016	2017	2018
province				
NL	Newfoundland and Labrador	724	734	750
PE	Prince Edward Island	584	601	638
NS	Nova Scotia	3060	3138	3201
NB	New Brunswick	2741	2814	2956
QC	Quebec	21372	22720	23749

In [40]:
```
# last 5 rows
prov_support.tail()
```

Out[40]:

	province_name	2016	2017	2018
province				
AB	Alberta	5772	5943	6157
BC	British Columbia	6482	6680	6925
YT	Yukon	946	973	1006
NT	Northwest Territories	1281	1294	1319
NU	Nunavut	1539	1583	1634

In [41]:
```
# It is possible to specify how many rows of data we want to see:

prov_support.head(3)
```

Out[41]:

	province_name	2016	2017	2018
province				
NL	Newfoundland and Labrador	724	734	750
PE	Prince Edward Island	584	601	638
NS	Nova Scotia	3060	3138	3201

Axes and indexes

To see the dimensions of the `DataFrame`, we can use the method `shape`, it returns a tuple in the form:

(number of rows, number of columns)

In [42]:
```
prov_support.shape
```

Out[42]: (13, 4)

The `columns` method returns a list of all columns in a `DataFrame`, and the `index` method returns a `DataFrame`'s index:

In [43]:
```
prov_support.columns
```

Out[43]:
```
Index(['province_name', '2016', '2017', '2018'], dtype='object')
```

In [44]:
```
prov_support.index
```

Out[44]:
```
Index(['NL', 'PE', 'NS', 'NB', 'QC', 'ON', 'MB', 'SK', 'AB', 'BC', 'YT', 'NT',
       'NU'],
      dtype='object', name='province')
```

An important concept used in `pandas` is **`DataFrame` axes**. In a `DataFrame`, axes are indexes where `axis=0` is the rows index, and `axis=1` is the full set of column names.

The `DataFrame` definition of axes is the same as the corresponding definition of axes in NumPy, which is: "A 2-dimensional array has two corresponding axes: the first running vertically downwards across rows (axis 0), and the second running horizontally across columns (axis 1)."

DataFrame axes

Picture 9. DataFrame axes.

DataFrame's method `axes` can be used to output both axes, the rows index (`axis=0`) first, then column names (`axis=1`).

In [45]:
```
prov_support.axes
```

Out[45]:
```
[Index(['NL', 'PE', 'NS', 'NB', 'QC', 'ON', 'MB', 'SK', 'AB', 'BC', 'YT', 'NT',
        'NU'],
       dtype='object', name='province'),
 Index(['province_name', '2016', '2017', '2018'], dtype='object')]
```

Multiple methods in `pandas` use `axis` as a parameter. Let's see an example. Pandas has the method `sort_index()` which will sort a `DataFrame` based on either index or column headers.

The code below will sort a `DataFrame` based on the index (`axis=0`). The default order is ascending — which means, in our example, alphabetical order since our index is of a *string* type. We can change the sort order using the `ascending=` parameter.

In [46]:
```
prov_support.sort_index(axis=0, ascending=True)
```

Out[46]:

province	province_name	2016	2017	2018
AB	Alberta	5772	5943	6157
BC	British Columbia	6482	6680	6925
MB	Manitoba	3531	3675	3965
NB	New Brunswick	2741	2814	2956
NL	Newfoundland and Labrador	724	734	750
NS	Nova Scotia	3060	3138	3201
NT	Northwest Territories	1281	1294	1319
NU	Nunavut	1539	1583	1634
ON	Ontario	21347	21101	21420
PE	Prince Edward Island	584	601	638
QC	Quebec	21372	22720	23749
SK	Saskatchewan	1565	1613	1673
YT	Yukon	946	973	1006

In [47]:
```
# Sorting by column names:

prov_support.sort_index(axis=1, ascending=True)
```

Out[47]:

province	2016	2017	2018	province_name
NL	724	734	750	Newfoundland and Labrador
PE	584	601	638	Prince Edward Island
NS	3060	3138	3201	Nova Scotia
NB	2741	2814	2956	New Brunswick
QC	21372	22720	23749	Quebec
ON	21347	21101	21420	Ontario
MB	3531	3675	3965	Manitoba
SK	1565	1613	1673	Saskatchewan
AB	5772	5943	6157	Alberta
BC	6482	6680	6925	British Columbia
YT	946	973	1006	Yukon
NT	1281	1294	1319	Northwest Territories
NU	1539	1583	1634	Nunavut

NOTE: `Series` have only one axis and it is a row axis.

You can also see that indexes are of `pandas` **`Index`** object type:

In [48]:
```
obj1 = prov_support.index
type(obj1)
```

Out[48]:
```
pandas.core.indexes.base.Index
```

In [49]:
```
obj2 = prov_support.columns
type(obj2)
```

Out[49]:
```
pandas.core.indexes.base.Index
```

In [50]:
```
print(obj1); print(obj2)
```

```
Index(['NL', 'PE', 'NS', 'NB', 'QC', 'ON', 'MB', 'SK', 'AB', 'BC', 'YT', 'NT',
       'NU'],
      dtype='object', name='province')
Index(['province_name', '2016', '2017', '2018'], dtype='object')
```

Hierarchical indexes in DataFrames

Pandas supports hierarchical indexes, or a multi-level index. Hierarchical indexes allows us to manipulate the data with any number of dimensions within a two-dimensional `DataFrame`. Please refer to `pandas` documentation for more details. We will come back to hierarchical indexes in Part 3 of this module, where we are going to talk about aggregating data in the `DataFrame`.

To introduce the concept, here is a simple example:

In [51]:
```
df_hierarch = pd.DataFrame(data=[4, 7, 2, 5, 6],
                           columns=['Data'],
                           index=
                           [['a', 'a', 'b', 'b', 'a'],
                            ['x', 'y', 'x', 'y', 'x']])
```

In [52]:
```
df_hierarch
```

Out[52]:

		Data
a	**x**	4
	y	7
b	**x**	2
	y	5
a	**x**	6

In [53]:
```
df_hierarch.index
```

Out[53]:
```
MultiIndex(levels=[['a', 'b'], ['x', 'y']],
           codes=[[0, 0, 1, 1, 0], [0, 1, 0, 1, 0]])
```

As you can see, the index of this `DataFrame` has 2 levels, one level has indexes `'a'` and `'b'`, and the second level, has `'x'` and `'y'`. The type of this index is `MultiIndex`.

Now that we understand axes and indexes, let's talk about selecting data, indexing and slicing.

Selecting data from a DataFrame, indexing and slicing

A column in a `DataFrame` can be retrieved either by dictionary-like notation using the column's name, or by attribute:

In [54]:
```
# Here we are using a column name

prov_support['province_name']
```

Out[54]:
```
province
NL    Newfoundland and Labrador
PE         Prince Edward Island
NS                  Nova Scotia
NB                New Brunswick
QC                       Quebec
ON                      Ontario
MB                     Manitoba
SK                 Saskatchewan
AB                      Alberta
BC             British Columbia
YT                        Yukon
NT        Northwest Territories
NU                      Nunavut
Name: province_name, dtype: object
```

In [55]:
```python
# Here we are using an attribute

prov_support.province_name
```

Out[55]:
```
province
NL    Newfoundland and Labrador
PE         Prince Edward Island
NS                  Nova Scotia
NB                New Brunswick
QC                       Quebec
ON                      Ontario
MB                     Manitoba
SK                 Saskatchewan
AB                      Alberta
BC             British Columbia
YT                        Yukon
NT        Northwest Territories
NU                      Nunavut
Name: province_name, dtype: object
```

The result is the same, in both cases the data is retrieved as a `Series` object, and in both cases it is exactly the same object. We can confirm it as follows:

In [56]:
```python
prov_support['province_name'] is prov_support.province_name
```

Out[56]:
```
True
```

In [57]:
```python
print(type(prov_support['province_name']))
print(type(prov_support.province_name))
```

```
<class 'pandas.core.series.Series'>
<class 'pandas.core.series.Series'>
```

NOTE: Retrieving data by column name, e.g. `prov_support['province_name']`, works for any column name, but getting the data by attribute will work only if the column name does not contain any spaces and adheres to Python variable naming rules. For example, this restriction means that we cannot use this approach to access columns which are labelled as years. Years are numbers, and Python variables cannot start with a number:

In [58]:
```
# This will return an error because a number is not a valid Python variable name

prov_support.2016
```

```
  File "<ipython-input-58-f161fe1197c2>", line 4
    prov_support.2016
                    ^
SyntaxError: invalid syntax
```

In [59]:
```
# This will always work:

prov_support['2016']
```

Out[59]:
```
province
NL      724
PE      584
NS     3060
NB     2741
QC    21372
ON    21347
MB     3531
SK     1565
AB     5772
BC     6482
YT      946
NT     1281
NU     1539
Name: 2016, dtype: int64
```

To select multiple columns of data, we need to pass a list of columns. The output will be a `DataFrame`:

In [60]:
```
prov_support[['province_name','2018']]
```

Out[60]:

province	province_name	2018
NL	Newfoundland and Labrador	750
PE	Prince Edward Island	638
NS	Nova Scotia	3201
NB	New Brunswick	2956
QC	Quebec	23749
ON	Ontario	21420
MB	Manitoba	3965
SK	Saskatchewan	1673
AB	Alberta	6157
BC	British Columbia	6925
YT	Yukon	1006
NT	Northwest Territories	1319
NU	Nunavut	1634

To select rows from the `DataFrame`, use the indexing operators `.loc` or `.iloc`:

- `.loc` is used to select rows by index labels
- `.iloc` is used to select rows by integer index, by position

Here are examples of both methods:

In [61]:
```
# Select data for Ontario by the label 'ON':
prov_support.loc['ON']
```

Out[61]:
```
province_name    Ontario
2016               21347
2017               21101
2018               21420
Name: ON, dtype: object
```

In [62]:
```
# Selecting data for Ontario, British Columbia and Quebec by labels:

prov_support.loc[['ON', 'BC', 'QC']]
```

Out[62]:

province	province_name	2016	2017	2018
ON	Ontario	21347	21101	21420
BC	British Columbia	6482	6680	6925
QC	Quebec	21372	22720	23749

In [63]:
```
# Selecting Ontario by position using .iloc method:

prov_support.iloc[5]
```

Out[63]:
```
province_name    Ontario
2016               21347
2017               21101
2018               21420
Name: ON, dtype: object
```

In [64]:
```
# Selecting data for Ontario, British Columbia and Quebec by position:

prov_support.iloc[[5,9,4]]
```

Out[64]:

province	province_name	2016	2017	2018
ON	Ontario	21347	21101	21420
BC	British Columbia	6482	6680	6925
QC	Quebec	21372	22720	23749

We can select a range, or a slice, of rows by specifying the start and end row position or index label. If we use row position and call `.iloc[start:end]`, then the row identified by the `end` position is not included in the resulting output.

In [65]:
```
prov_support.iloc[:3]   # rows with position 0, 1 and 2, but not 3
```

Out[65]:

province	province_name	2016	2017	2018
NL	Newfoundland and Labrador	724	734	750
PE	Prince Edward Island	584	601	638
NS	Nova Scotia	3060	3138	3201

However, when we use index labels, both the start and the end positions are included:

In [66]:
```
prov_support.loc[:'NS']
```

Out[66]:

province	province_name	2016	2017	2018
NL	Newfoundland and Labrador	724	734	750
PE	Prince Edward Island	584	601	638
NS	Nova Scotia	3060	3138	3201

In [67]:

```
# Only numeric columns:

prov_support[['2016', '2017', '2018']]
```

Out[67]:

province	2016	2017	2018
NL	724	734	750
PE	584	601	638
NS	3060	3138	3201
NB	2741	2814	2956
QC	21372	22720	23749
ON	21347	21101	21420
MB	3531	3675	3965
SK	1565	1613	1673
AB	5772	5943	6157
BC	6482	6680	6925
YT	946	973	1006
NT	1281	1294	1319
NU	1539	1583	1634

In [68]:

```
# We can achieve the same result using .iloc()

prov_support.iloc[:, 1:]
```

Out[68]:

	2016	2017	2018
province			
NL	724	734	750
PE	584	601	638
NS	3060	3138	3201
NB	2741	2814	2956
QC	21372	22720	23749
ON	21347	21101	21420
MB	3531	3675	3965
SK	1565	1613	1673
AB	5772	5943	6157
BC	6482	6680	6925
YT	946	973	1006
NT	1281	1294	1319
NU	1539	1583	1634

As we just saw, we can select a slice (subset) of rows and columns. In the example below, we are selecting the first 3 rows and the first 2 columns only:

In [69]:
```
prov_support.iloc[:3, :2]
```

Out[69]:

	province_name	2016
province		
NL	Newfoundland and Labrador	724
PE	Prince Edward Island	584
NS	Nova Scotia	3060

In [70]:
```
# Same, but using row and column labels

prov_support.loc[:'NS', :'2016']
```

Out[70]:

province	province_name	2016
NL	Newfoundland and Labrador	724
PE	Prince Edward Island	584
NS	Nova Scotia	3060

Sometimes there is a need to select data based on a condition, for example, select all rows where federal support in 2018 is greater than 3000. This is called **Boolean indexing**:

In [71]:
```
prov_support[prov_support['2018'] > 3000]
```

Out[71]:

province	province_name	2016	2017	2018
NS	Nova Scotia	3060	3138	3201
QC	Quebec	21372	22720	23749
ON	Ontario	21347	21101	21420
MB	Manitoba	3531	3675	3965
AB	Alberta	5772	5943	6157
BC	British Columbia	6482	6680	6925

If there is a list of values that we are looking for within the `DataFrame`, we can use `isin()` function. In the example below, we are looking for rows with indexes 'AB', 'ON' and 'BC':

In [72]:
```
prov_support[prov_support.index.isin(['AB', 'ON', 'BC'])]
```

Out[72]:

province	province_name	2016	2017	2018
ON	Ontario	21347	21101	21420
AB	Alberta	5772	5943	6157
BC	British Columbia	6482	6680	6925

In [73]:
```
prov_support[prov_support['2017'].isin(['22720', '21101'])]
```

Out[73]:

province	province_name	2016	2017	2018
QC	Quebec	21372	22720	23749
ON	Ontario	21347	21101	21420

If we need to combine multiple conditions, we can use the `&` operator:

In [74]:
```
prov_support[(prov_support['2017'] > 6000) & (prov_support.index.isin(['AB', 'ON',
```

Out[74]:

province	province_name	2016	2017	2018
ON	Ontario	21347	21101	21420
BC	British Columbia	6482	6680	6925

"Is not" can be coded using the `~` unary operator:

In [75]:
```
# Selecting rows where support in 2017 does not equal to 6680:
prov_support[~(prov_support['2017'] == 6680)]  # returns all rows except British Co
```

Out[75]:

province	province_name	2016	2017	2018
NL	Newfoundland and Labrador	724	734	750
PE	Prince Edward Island	584	601	638
NS	Nova Scotia	3060	3138	3201
NB	New Brunswick	2741	2814	2956
QC	Quebec	21372	22720	23749
ON	Ontario	21347	21101	21420
MB	Manitoba	3531	3675	3965
SK	Saskatchewan	1565	1613	1673
AB	Alberta	5772	5943	6157
YT	Yukon	946	973	1006
NT	Northwest Territories	1281	1294	1319
NU	Nunavut	1539	1583	1634

In [76]:
```
# Another way of doing the same thing:

prov_support[prov_support['2017'] != 6680]
```

Out[76]:

province	province_name	2016	2017	2018
NL	Newfoundland and Labrador	724	734	750
PE	Prince Edward Island	584	601	638
NS	Nova Scotia	3060	3138	3201
NB	New Brunswick	2741	2814	2956
QC	Quebec	21372	22720	23749
ON	Ontario	21347	21101	21420
MB	Manitoba	3531	3675	3965
SK	Saskatchewan	1565	1613	1673
AB	Alberta	5772	5943	6157
YT	Yukon	946	973	1006
NT	Northwest Territories	1281	1294	1319
NU	Nunavut	1539	1583	1634

Pandas string methods

We can use all `string` methods in `pandas` `DataFrame`s via the `str()` method. For example, here is how we can convert all province names to lower case:

In [77]:

```
prov_support.province_name.str.lower()
```

Out[77]:

```
province
NL    newfoundland and labrador
PE         prince edward island
NS                  nova scotia
NB                 new brunswick
QC                       quebec
ON                      ontario
MB                     manitoba
SK                 saskatchewan
AB                      alberta
BC             british columbia
YT                        yukon
NT        northwest territories
NU                      nunavut
Name: province_name, dtype: object
```

In [78]:

```
# Selecting all rows where province name contains the character 'm'
prov_support[prov_support.province_name.str.contains('m')]
```

Out[78]:

	province_name	2016	2017	2018
province				
BC	British Columbia	6482	6680	6925

In [79]:
```
'''In the result set, we don't see Manitoba because the name of the province
   starts with an upper-case 'M'. Let's fix it: '''

prov_support[prov_support.province_name.str.lower().str.contains('m')]
```

Out[79]:

	province_name	2016	2017	2018
province				
MB	Manitoba	3531	3675	3965
BC	British Columbia	6482	6680	6925

In [80]:
```
# Same result as above, but we want only a column with province names:
prov_support.loc[prov_support.province_name.str.lower().str.contains('m'), ['provin
```

Out[80]:

	province_name
province	
MB	Manitoba
BC	British Columbia

NOTE: We can use all built-in Python string methods in `pandas`. Their application is not limited to the methods we used in this section.

EXERCISE 4: Loading data into DataFrame, exploring

For this exercise, we will use the Chocolate Bar Ratings dataset. We have downloaded the data from Kaggle, "Data" tab. The dataset contains expert ratings of over 1,700 individual chocolate bars, along with information on their regional origin, percentage of cocoa, the variety of chocolate bean used and where the beans were grown.

You can read about the data either on the "Overview" page of the dataset on Kaggle web site, or go to the Flavors of Cacao website where the data originally came from, and read about the rating system used in this dataset, overview of factors that contribute to the chocolate flavour and other useful and interesting information about chocolate.

The dataset contains the following data attributes:

Attribute	*Description*
Company (Maker-if known)	Name of the company manufacturing the bar
Specific Bean Origin or Bar Name	The specific geo-region of origin for the bar
REF	Reference number, a value linked to when the review was entered in the database. Higher = more recent.
Review Date	Date of publication of the review
Cocoa Percent	Cocoa percentage (darkness) of the chocolate bar being reviewed
Company Location	Manufacturer base country
Rating	Expert rating for the bar
Bean Type	The variety of bean used, if provided
Broad Bean Origin	The broad geo-region of origin for the bean

Task 1. Load the data from the `flavors_of_cacao.csv` file into a `DataFrame`. If the file is in different directory on your system, make sure to include the full path to the file.

Answer the following questions:

- How many entries are there in the dataset?
- What is the `dtype` of the data in the `Rating` column?

In [43]:
```
import pandas as pd

cacao = pd.read_csv('flavors_of_cacao.csv',skiprows=1,   # skipping one row
                                           header=None,  # we are telling pandas th
                                           names=['Company','Origin','REF','Review_

cacao
```

Out[43]:

	Company	Origin	REF	Review_Date	Cocao %	Location	Rating	Type	Geo-Origin
0	A. Morin	Agua Grande	1876	2016	63%	France	3.75		Sao Tome
1	A. Morin	Kpime	1676	2015	70%	France	2.75		Togo
2	A. Morin	Atsane	1676	2015	70%	France	3.00		Togo
3	A. Morin	Akata	1680	2015	70%	France	3.50		Togo
4	A. Morin	Quilla	1704	2015	70%	France	3.50		Peru
...	...	...	...	...	...	...	...	...	...
1790	Zotter	Peru	647	2011	70%	Austria	3.75		Peru
1791	Zotter	Congo	749	2011	65%	Austria	3.00	Forastero	Congo
1792	Zotter	Kerala State	749	2011	65%	Austria	3.50	Forastero	India
1793	Zotter	Kerala State	781	2011	62%	Austria	3.25		India
1794	Zotter	Brazil, Mitzi Blue	486	2010	65%	Austria	3.00		Brazil

1795 rows × 9 columns

In [44]:
```
cacao.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1795 entries, 0 to 1794
Data columns (total 9 columns):
 #   Column       Non-Null Count  Dtype  
---  ------       --------------  -----  
 0   Company      1795 non-null   object 
 1   Origin       1795 non-null   object 
 2   REF          1795 non-null   int64  
 3   Review_Date  1795 non-null   int64  
 4   Cocao %      1795 non-null   object 
 5   Location     1795 non-null   object 
 6   Rating       1795 non-null   float64
 7   Type         1794 non-null   object 
 8   Geo-Origin   1794 non-null   object 
dtypes: float64(1), int64(2), object(6)
memory usage: 126.3+ KB
```

In [50]:
```
(cacao['Rating']).dtypes
```

Out[50]:
```
dtype('float64')
```

Task 2. In this task, we want to take a look at the header row of the `DataFrame` with column names. We also want to investigate the "Bean Type" column.

- Print out the column names. What can you tell about the format of the column names?
- Update column names, assign names that are easy to work with. You can update all column names or only some of them, and you can assign new names.
- Explore the "Bean Type" column. How many entries are there in this column? Are there any empty values?

In [52]:
```
cacao.columns
```

Out[52]:
```
Index(['Company', 'Origin', 'REF', 'Review_Date', 'Cocao %', 'Location',
       'Rating', 'Type', 'Geo-Origin'],
      dtype='object')
```

In [54]:
```
cacao.Type[2]
```

Out[54]:
```
'\xa0'
```

In [56]:
```
import numpy as np

cacao.replace(u'\xa0', np.nan, regex=True, inplace=True)
```

In [57]:
```
cacao.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1795 entries, 0 to 1794
Data columns (total 9 columns):
 #   Column       Non-Null Count  Dtype  
---  ------       --------------  -----  
 0   Company      1795 non-null   object 
 1   Origin       1795 non-null   object 
 2   REF          1795 non-null   int64  
 3   Review_Date  1795 non-null   int64  
 4   Cocao %      1795 non-null   object 
 5   Location     1795 non-null   object 
 6   Rating       1795 non-null   float64
 7   Type         907 non-null    object 
 8   Geo-Origin   1721 non-null   object 
dtypes: float64(1), int64(2), object(6)
memory usage: 126.3+ KB
```

In [ ]:

Solutions

Task 1.

In [81]:
```
# We can simply load all the data without specifying any parameters at this point

chocolate = pd.read_csv('flavors_of_cacao.csv')
chocolate.head(10)
```

Out[81]:

	Company (Maker-if known)	Specific Bean Origin or Bar Name	REF	Review Date	Cocoa Percent	Company Location	Rating	Bean Type	Broad Bean Origin
0	A. Morin	Agua Grande	1876	2016	63%	France	3.75		Sao Tome
1	A. Morin	Kpime	1676	2015	70%	France	2.75		Togo
2	A. Morin	Atsane	1676	2015	70%	France	3.00		Togo
3	A. Morin	Akata	1680	2015	70%	France	3.50		Togo
4	A. Morin	Quilla	1704	2015	70%	France	3.50		Peru
5	A. Morin	Carenero	1315	2014	70%	France	2.75	Criollo	Venezuela
6	A. Morin	Cuba	1315	2014	70%	France	3.50		Cuba
7	A. Morin	Sur del Lago	1315	2014	70%	France	3.50	Criollo	Venezuela
8	A. Morin	Puerto Cabello	1319	2014	70%	France	3.75	Criollo	Venezuela
9	A. Morin	Pablino	1319	2014	70%	France	4.00		Peru

In order to answer the questions, we can use the `info()` method to explore the dataset:

In [82]:
```
chocolate.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1795 entries, 0 to 1794
Data columns (total 9 columns):
Company 
(Maker-if known)           1795 non-null object
Specific Bean Origin
or Bar Name    1795 non-null object
REF                                 1795 non-null int64
Review
Date                         1795 non-null int64
Cocoa
Percent                       1795 non-null object
Company
Location                     1795 non-null object
Rating                              1795 non-null float64
Bean
Type                            1794 non-null object
Broad Bean
Origin                    1794 non-null object
dtypes: float64(1), int64(2), object(6)
memory usage: 126.3+ KB
```

In [83]:
```
# We can also use `.shape` attribute
# it will output 2 numbers, the number of rows and number of columns
```

```
# this will give us an answer to the first question only

chocolate.shape
```

Out[83]: (1795, 9)

Q1: How many entries are there in the dataset?

Answer: There are 1795 entries in the dataset, we can see it from the second line:

```
RangeIndex: 1795 entries, 0 to 1794
```

Q2: What is the `dtype` of the data in the `Rating` column?

Answer: we can answer this question by looking at the output of `info()` above. The `Rating` column has `float64` dtype. We can also retrieve information about this column as follows:

In [84]:
```
chocolate[['Rating']].info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1795 entries, 0 to 1794
Data columns (total 1 columns):
Rating    1795 non-null float64
dtypes: float64(1)
memory usage: 14.1 KB
```

In [85]:
```
(chocolate['Rating']).dtypes
```

Out[85]: dtype('float64')

Task 2. Let's print out the names of the columns:

In [86]:
```
chocolate.columns
```

Out[86]:
```
Index(['Company \n(Maker-if known)', 'Specific Bean Origin\nor Bar Name',
       'REF', 'Review\nDate', 'Cocoa\nPercent', 'Company\nLocation', 'Rating',
       'Bean\nType', 'Broad Bean\nOrigin'],
      dtype='object')
```

You can see that almost every column name contains a special character, `\n`, which is a newline character. Also, some of the column names are quite long, e.g. `'Company \n(Maker-if known)'`. To simplify data manipulation, we suggest renaming the columns:

In [87]:
```
chocolate.columns = ['company', 'bar_name', 'REF', 'review_date',
                     'cocoa_per', 'company_location', 'rating','bean_type', 'bean_o
chocolate.head(10)
```

Out[87]:

	company	bar_name	REF	review_date	cocoa_per	company_location	rating	bean_typ
0	A. Morin	Agua Grande	1876	2016	63%	France	3.75	
1	A. Morin	Kpime	1676	2015	70%	France	2.75	
2	A. Morin	Atsane	1676	2015	70%	France	3.00	
3	A. Morin	Akata	1680	2015	70%	France	3.50	
4	A. Morin	Quilla	1704	2015	70%	France	3.50	
5	A. Morin	Carenero	1315	2014	70%	France	2.75	Crioll
6	A. Morin	Cuba	1315	2014	70%	France	3.50	
7	A. Morin	Sur del Lago	1315	2014	70%	France	3.50	Crioll
8	A. Morin	Puerto Cabello	1319	2014	70%	France	3.75	Crioll
9	A. Morin	Pablino	1319	2014	70%	France	4.00	

The answer to the last question requires a little investigation. If you look at the output of the `info()` function, you will notice that there are 1974 entries in this column, but in the output of the `head(10)` function we can see that only 3 out of 10 rows have values populated. This means that other rows contain some special characters which `pandas` treat as values. Let's pick a row and look closer at the `bean_type` value:

In [88]:
```
chocolate.bean_type[2]
```

Out[88]: '\xa0'

The `\xa0` is a symbol for non-breaking space. We can check unique values in this column and see if there are other special characters in the data:

In [89]:
```
chocolate.bean_type.unique()
```

```
Out[89]: array(['\xa0', 'Criollo', 'Trinitario', 'Forastero (Arriba)', 'Forastero',
       'Forastero (Nacional)', 'Criollo, Trinitario',
       'Criollo (Porcelana)', 'Blend', 'Trinitario (85% Criollo)',
       'Forastero (Catongo)', 'Forastero (Parazinho)',
       'Trinitario, Criollo', 'CCN51', 'Criollo (Ocumare)', 'Nacional',
       'Criollo (Ocumare 61)', 'Criollo (Ocumare 77)',
       'Criollo (Ocumare 67)', 'Criollo (Wild)', 'Beniano', 'Amazon mix',
       'Trinitario, Forastero', 'Forastero (Arriba) ASS', 'Criollo, +',
       'Amazon', 'Amazon, ICS', 'EET', 'Blend-Forastero,Criollo',
       'Trinitario (Scavina)', 'Criollo, Forastero', 'Matina',
       'Forastero(Arriba, CCN)', 'Nacional (Arriba)',
       'Forastero (Arriba) ASSS', 'Forastero, Trinitario',
       'Forastero (Amelonado)', nan, 'Trinitario, Nacional',
       'Trinitario (Amelonado)', 'Trinitario, TCGA', 'Criollo (Amarru)'],
      dtype=object)
```

Starting from Module 5, we will cover how to deal with the empty/missing values and how to clean data and prepare it for further analysis. For now, we will provide you with a quick solution to clean up the `bean_type` column:

```
In [90]: import numpy as np

chocolate.replace(u'\xa0', np.nan, regex=True, inplace=True)
chocolate.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1795 entries, 0 to 1794
Data columns (total 9 columns):
company             1795 non-null object
bar_name            1795 non-null object
REF                 1795 non-null int64
review_date         1795 non-null int64
cocoa_per           1795 non-null object
company_location    1795 non-null object
rating              1795 non-null float64
bean_type           907 non-null object
bean_origin         1721 non-null object
dtypes: float64(1), int64(2), object(6)
memory usage: 126.3+ KB
```

We simply replaced the `'\xa0'` character with `NaN` values. As you can see, now the `bean_type` column contains only 907 values.

NOTE: The `u` character in front of the `'\xa0'` indicates that we're asking Python to treat this character as a Unicode string. For more information on Unicode please refer to the Python Documentation Unicode HOWTO.

Another way to clean up the non-breaking spaces from the dataset is to pass `na_values='\xa0'` as a parameter to the `read_csv()` function and remove these special characters at the time we read the `.csv` file and create the `DataFrame`:

```
In [91]: chocolate = pd.read_csv('flavors_of_cacao.csv', na_values='\xa0')
chocolate.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1795 entries, 0 to 1794
Data columns (total 9 columns):
Company
(Maker-if known)           1795 non-null object
Specific Bean Origin
or Bar Name    1795 non-null object
REF                                 1795 non-null int64
Review
Date                         1795 non-null int64
Cocoa
Percent                       1795 non-null object
Company
Location                     1795 non-null object
Rating                              1795 non-null float64
Bean
Type                           907 non-null object
Broad Bean
Origin                    1721 non-null object
dtypes: float64(1), int64(2), object(6)
memory usage: 126.3+ KB
```

You will learn more about handling Null values in the next module.

End of Part 2

This notebook makes up one part of this module. Now that you have completed this part, please proceed to the next notebook in this module.

If you have any questions, please reach out to your peers using the discussion boards. If you and your peers are unable to come to a suitable conclusion, do not hesitate to reach out to your instructor on the designated discussion board.

References

Pandas 0.23.4 documentation. API Reference. (2018). `pandas.Series`. Retrieved from https://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.html

Statistics Canada. (2018). Canada at a Glance 2018. Population. Retrieved from https://www150.statcan.gc.ca/n1/pub/12-581-x/2018000/pop-eng.htm

Pandas 0.23.4 documentation. API Reference. (2018). `pandas.Series.append`. Retrieved from https://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.append.html

Wikipedia. (2018). Provinces and territories of Canada. Retrieved from https://en.wikipedia.org/wiki/Provinces_and_territories_of_Canada

Department of Finance Canada. (2018). Federal Support to Provinces and Territories, 2018-19. Retrieved from https://www.fin.gc.ca/fedprov/mtp-eng.asp

Pandas 0.23.4 documentation. API Reference. (2018). `pandas.read_csv`. Retrieved from https://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html

SciPy Community at SciPy.org. (2018). Glossary, topic "along an axis". Retrieved from https://docs.scipy.org/doc/numpy/glossary.html

Pandas 0.23.4 documentation. (2018). MultiIndex / Advanced Indexing. Retrieved from https://pandas.pydata.org/pandas-docs/stable/advanced.html

Kaggle. (2017). Chocolate Bar Ratings dataset. Retrieved from https://www.kaggle.com/rtatman/chocolate-bar-ratings

Flavors of Cacao (2018). Website URL is http://flavorsofcacao.com/index.html

Wikipedia (2018). Non-breaking space. Retrieved from https://en.wikipedia.org/wiki/Non-breaking_space

Python Documentation. Python HOWTOs. (2018). Unicode HOWTO. Retrieved from https://docs.python.org/3/howto/unicode.html.