Data Scientist Path - Python

Dataquest

By

Daniel Ruiz

December 2019

Prelude

This set of notes tracks my journey through DataQuests' *Data Scientist Path* module with Python. These notes are one of many that I have decided to polish and make available for anyone interested. One of the benefits of this series are having a compendium of definitions, examples and personal thoughts that I can always refer back to if I need a reminder on a particular topic. In addition, I find that this medium reduces the search time for specific definitions, theorems, examples etc and thus aids in reinforcing my own knowledge when frequented. The act of writing also works as a ritual in helping to encode information within my brain.

In these notes, I have provided many definitions and examples that I have encountered through the *Data Scientist* module on Dataquest. We will now outline their brief summaries:

- §1: **Data Analysis and Visualization** goes over the fundamentals of the Python Pandas and NumPy modules. I also provide details on Data Exploration and Visualization and how to do this in Python.

Contents

1 Data Analysis and Visualization

1.1 Pandas and NumPy Fundamentals

1.1.1 Introduction to NumPy

- CSV: Open, Read etc
- NdArray.shape
- Selecting rows/columns from an array
- NdArray Arithmetic: Addition, Subtraction, Multiplication, Division
- NdArray Methods: Mean, min, max, sum
- Overlap with Numpy Functions and Methods

Definition 1.1: NumPy Library

NumPy is the fundamental package for scientific computing with Python. It contains among other things:

- *A powerful N-dimensional array object.*
- *Sophisticated (broadcasting) functions.*
- *Tools for integrating C/C++ and Fortran code.*
- *Useful linear algebra, Fourier transform, and random number capabilities*

Besides its obvious scientific uses, NumPy can also be used as an efficient multi-dimensional container of generic data. Arbitrary data-types can be defined. This allows NumPy to seamlessly and speedily integrate with a wide variety of databases.

Definition 1.2: NdArray

```
class numpy.ndarray(shape, dtype=float, buffer=None, offset=0, strides=None,
order=None)
```

An array object represents a multidimensional, homogeneous array of fixed-size items. An associated data-type object describes the format of each element in the array (its byte-order, how many bytes it occupies in memory, whether it is an integer, a floating point number, or something else, etc.)

Arrays should be constructed using array, zeros or empty (refer to the See Also section below). The parameters given here refer to a low-level method (ndarray(. . .)) for instantiating an array.

PARAMETERS
shape *: tuple of ints*
Shape of created array.

dtype *: data-type, optional*
Any object that can be interpreted as a numpy data type.

buffer*: object exposing buffer interface, optional*
Used to fill the array with data.

offset*: int, optional*
Offset of array data in buffer.

strides *: tuple of ints, optional*
Strides of data in memory.

order: *{'C', 'F'}, optional*
Row-major (C-style) or column-major (Fortran-style) order.

Definition 1.3: SIMD

Single instruction, multiple data *(SIMD) is a class of parallel computers in Flynn's taxonomy. It describes computers with multiple processing elements that perform the same operation on multiple data points simultaneously. Such machines exploit data level parallelism, but not concurrency: there are simultaneous (parallel) computations, but only a single process (instruction) at a given moment.*

Definition 1.4: Slice Object

class slice(start, stop, step)

Return a slice object representing the set of indices specified by range(start, stop, step). The start and step arguments default to None. Slice objects have read-only data attributes start, stop and step which merely return the argument values (or their default).

Ndarray Indexing

Ndarrays can be indexed using the standard Python x[obj] syntax, where x is the array and obj the selection. There are three kinds of indexing available: field access, basic slicing, advanced indexing. Which one occurs depends on obj.

Basic slicing extends Python's basic concept of slicing to N dimensions. Basic slicing occurs when obj is a slice object (constructed by start:stop:step notation inside of brackets), an integer, or a tuple of slice objects and integers. Ellipsis and newaxis objects can be interspersed with these as well.

1.1.2 Boolean Indexing with NumPy

- genfromtxt function
- NaN (Not a number)
- Boolean Arrays
- Boolean Indexing
- Assigning values to NdArrays with Indexing

Definition 1.5: Numpy genfromtxt()

```
numpy.genfromtxt(fname, dtype=<class 'float'>, comments='#', delimiter=None,
skip_header=0)
```

Load data from a text file, with missing values handled as specified.
Each line past the first skip_header lines is split at the delimiter character, and characters following the comments character are discarded. For further parameters and documentation, see [1].

PARAMETERS
fname: *file, str, pathlib.Path, list of str, generator.*
File, filename, list, or generator to read. If the filename extension is gz or bz2, the file is first decompressed. Note that generators must return byte strings in Python 3k. The strings in a list or produced by a generator

are treated as lines.

dtype*: dtype, optional.*
Data type of the resulting array. If None, the dtypes will be determined by the contents of each column, individually.

comments*: str, optional.*
The character used to indicate the start of a comment. All the characters occurring on a line after a comment are discarded.

delimiter*: str, int, or sequence, optional.*
The string used to separate values. By default, any consecutive whitespaces act as delimiter. An integer or sequence of integers can also be provided as width(s) of each field.

skip_header *: int, optional.*
The number of lines to skip at the beginning of the file.

Definition 1.6: Numpy nan

numpy.nan

Python constant. IEEE 754 floating point representation of Not a Number (NaN).
NaN and NAN are equivalent definitions of nan. Please use nan instead of NAN.

1.1.3 Introduction to Pandas

- Dataframe Methods: Head, Tail, Info, Loc
- Series object
- Selecting Columns/Rows in a Dataframe

Definition 1.7: Pandas Library

Pandas is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language. Some key features of the library include:

• *A fast and efficient DataFrame object for data manipulation with integrated indexing.*
• *Tools for reading and writing data between in-memory data structures and different formats: CSV and text files, Microsoft Excel, SQL databases, and the fast HDF5 format.*
• *Intelligent data alignment and integrated handling of missing data: gain automatic label-based alignment in computations and easily manipulate messy data into an orderly form.*

Definition 1.8: Dataframe

```
class pandas.DataFrame(data=None, index=None, columns=None, dtype=None,
copy=False)
```

Two-dimensional size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns). Arithmetic operations align on both row and column labels. Can be thought of as a dict-like container for Series objects. The primary pandas data structure. For a list of methods and further documentation, see [2].

PARAMETERS
data: *ndarray (structured or homogeneous), Iterable, dict, or DataFrame.*
Dict can contain Series, arrays, constants, or list-like objects.
Changed in version 0.23.0: If data is a dict, column order follows insertion-order for Python 3.6 and later.
Changed in version 0.25.0: If data is a list of dicts, column order follows insertion-order for Python 3.6 and later.

index: *Index or array-like.*
Index to use for resulting frame. Will default to RangeIndex if no indexing information part of input data and no index provided.

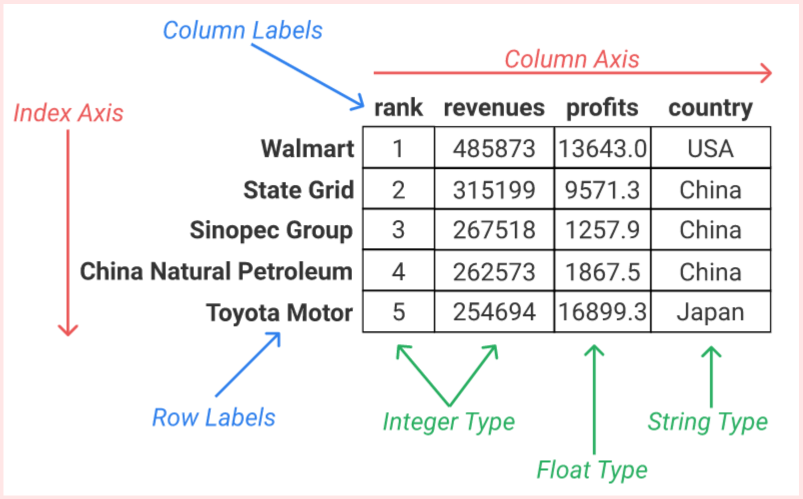
columns: *Index or array-like.*
Column labels to use for resulting frame. Will default to RangeIndex (0, 1, 2,.., n) if no column labels are provided.

dtype: *dtype, default None.*
Data type to force. Only a single dtype is allowed. If None, infer.

copy: *boolean, default False.*
Copy data from inputs. Only affects DataFrame / 2d ndarray input

Example 1.1: Pandas Dataframe

In vanilla numpy, our arrays must house elements of the same data type. Pandas extends this by allowing for multiple data types.



	rank	revenues	profits	country
Walmart	1	485873	13643.0	USA
State Grid	2	315199	9571.3	China
Sinopec Group	3	267518	1257.9	China
China Natural Petroleum	4	262573	1867.5	China
Toyota Motor	5	254694	16899.3	Japan

Figure 1.1: Pandas Dataframe

Definition 1.9: File-like Object (Pandas)

By file-like object, we refer to objects with a read() method, such as a file handler (e.g. via builtin open function) or StringIO.

Definition 1.10: Pandas read_csv()

```
pandas.read_csv(filepath_or_buffer, sep=',', delimiter=None, header='infer',
names=None, index_col=None)
```

Read a comma-separated values (csv) file into DataFrame. Also supports optionally iterating or breaking of the file into chunks. There are much more parameters available for this function, see documentation [3].

PARAMETERS
filepath_or_buffer *: str, path object or file-like object.*
Any valid string path is acceptable. The string could be a URL. Valid URL schemes include http, ftp, s3, and file. For file URLs, a host is expected. A local file could be: file://localhost/path/to/table.csv. If you want to pass in a path object, pandas accepts any os.PathLike.

sep*: str, default ‘,’.*
Delimiter to use. If sep is None, the C engine cannot automatically detect the separator, but the Python parsing engine can, meaning the latter will be used and automatically detect the separator by Python’s builtin sniffer tool, csv.Sniffer.

delimiter*: str, default None. Alias for sep.*

header*: int, list of int, default ‘infer’.*
Row number(s) to use as the column names, and the start of the data. Default behavior is to infer the column names: if no names are passed the behavior is identical to header=0 and column names are inferred from the first line of the file, if column names are passed explicitly then the behavior is identical to header=None. Explicitly pass header=0 to be able to replace existing names. The header can be a list of integers that specify row locations for a multi-index on the columns e.g. [0,1,3]. Intervening rows that are not specified will be skipped (e.g. 2 in this example is skipped). Note that this parameter ignores commented lines and empty lines if skip_blank_lines=True, so header=0 denotes the first line of data rather than the first line of the file.

names*: array-like, optional.*
List of column names to use. If file contains no header row, then you should explicitly pass header=None. Duplicates in this list are not allowed.

index_col*: int, str, sequence of int / str, or False, default None.*
Column(s) to use as the row labels of the DataFrame, either given as string name or column index. If a sequence of int / str is given, a MultiIndex is used.

Definition 1.11: Series

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

One-dimensional ndarray with axis labels (including time series).

Labels need not be unique but must be a hashable type. The object supports both integer- and label-based indexing and provides a host of methods for performing operations involving the index. Statistical methods from ndarray have been overridden to automatically exclude missing data (currently represented as NaN). For further methods and documentation, see [4].

*Operations between Series (+, -, /, *) align values based on their associated index values– they need not be the same length. The result index will be the sorted union of the two indexes.*

PARAMETERS
data: *array-like, Iterable, dict, or scalar value.*
Contains data stored in Series.
Changed in version 0.23.0: If data is a dict, argument order is maintained for Python 3.6 and later.

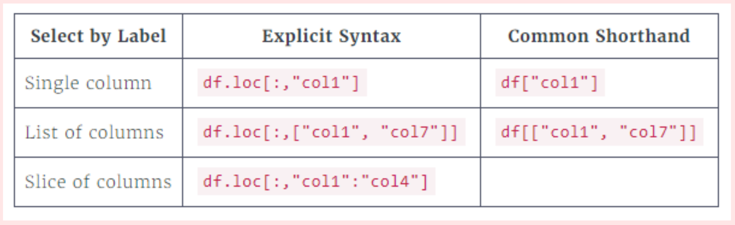
index*: array-like or Index (1d).*
Values must be hashable and have the same length as data. Non-unique index values are allowed. Will default to RangeIndex (0, 1, 2, .., n) if not provided. If both a dict and index sequence are used, the index will override the keys found in the dict.

dtype*: str, numpy.dtype, or ExtensionDtype, optional.*
Data type for the output Series. If not specified, this will be inferred from data. See the user guide for more usages.

copy*: bool, default False.*
Copy input data.

Example 1.2: Dataframe Indexing

Let df denote a Dataframe. Then, we can select some subset of the dataframe's columns via the following syntax:

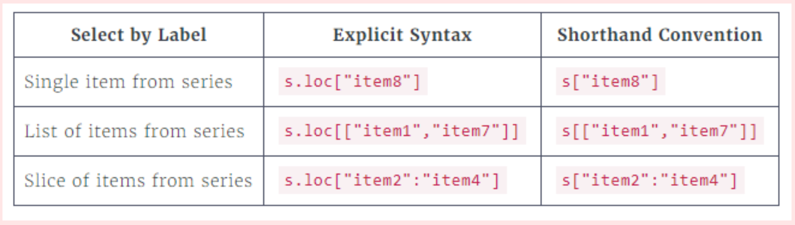


Select by Label	Explicit Syntax	Common Shorthand
Single column	`df.loc[:,"col1"]`	`df["col1"]`
List of columns	`df.loc[:,["col1", "col7"]]`	`df[["col1", "col7"]]`
Slice of columns	`df.loc[:,"col1":"col4"]`	

Figure 1.2: Indexing a Dataframe's Columns

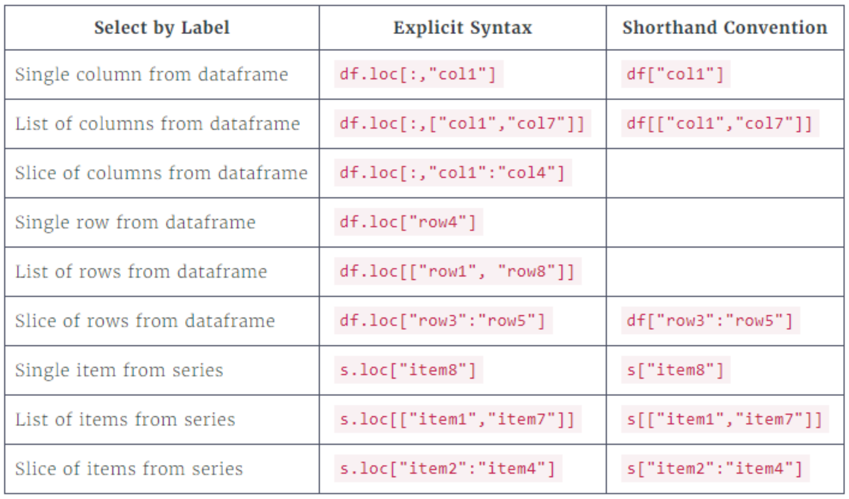
Example 1.3: Series Indexing

Let s denote a Series. Then, we can select some subset of the series rows via the following syntax:



Select by Label	Explicit Syntax	Shorthand Convention
Single item from series	`s.loc["item8"]`	`s["item8"]`
List of items from series	`s.loc[["item1","item7"]]`	`s[["item1","item7"]]`
Slice of items from series	`s.loc["item2":"item4"]`	`s["item2":"item4"]`

Figure 1.3: Series Item(s) Selection



Select by Label	Explicit Syntax	Shorthand Convention
Single column from dataframe	`df.loc[:,"col1"]`	`df["col1"]`
List of columns from dataframe	`df.loc[:,["col1","col7"]]`	`df[["col1","col7"]]`
Slice of columns from dataframe	`df.loc[:,"col1":"col4"]`	
Single row from dataframe	`df.loc["row4"]`	
List of rows from dataframe	`df.loc[["row1", "row8"]]`	
Slice of rows from dataframe	`df.loc["row3":"row5"]`	`df["row3":"row5"]`
Single item from series	`s.loc["item8"]`	`s["item8"]`
List of items from series	`s.loc[["item1","item7"]]`	`s[["item1","item7"]]`
Slice of items from series	`s.loc["item2":"item4"]`	`s["item2":"item4"]`

Figure 1.4: Summary of Dataframe / Series Item Selection

1.1.4 Exploring Data with Pandas: Fundamentals

- Dataframe Assignment
- Boolean Indexing with Dataframes
- New Columns in Dataframes

Pandas Arithmetic with Series

Because pandas is designed to operate like NumPy, a lot of concepts and methods from Numpy are supported. Recall that one of the ways NumPy makes working with data easier is with vectorized operations, or operations applied to multiple data points at once.

Just like with NumPy, we can use any of the standard Python numeric operators with series, including:

- *series_a + series_b: Addition*
- *series_a - series_b: Subtraction*
- *series_a * series_b: Multiplication (Element-wise)*
- *series_a / series_b: Division (Element-wise)*

Definition 1.12: Method Chaining

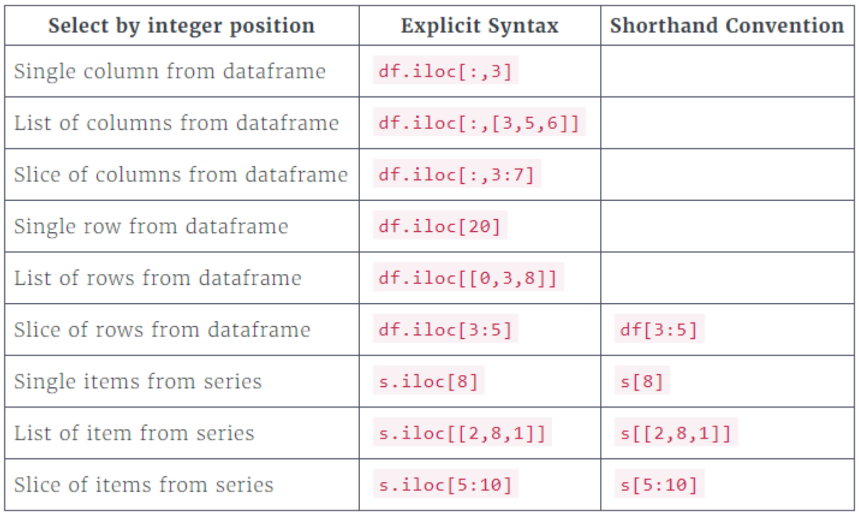
Method chaining, also known as named parameter idiom, is a common syntax for invoking multiple method calls in object-oriented programming languages. Each method returns an object, allowing the calls to be chained together in a single statement without requiring variables to store the intermediate results.

Useful Dataframe Methods

- ***df.describe()**: Generate descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding NaN values.*
- ***df.mean()**: Return the mean of the values for the requested axis.*
- ***df.rename()**: Alter axes labels. Function / dict values must be unique (1-to-1). Labels not contained in a dict / Series will be left as-is. Extra labels listed don't throw an error.*

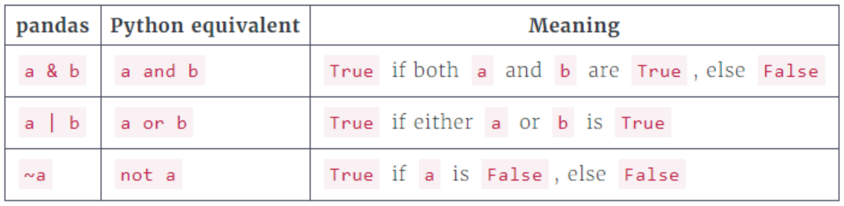
1.1.5 Exploring Data with Pandas: Intermediate

- Dataframe iloc indexing
- Boolean Operators; Combining Boolean Arrays
- Dataframe sort_values method



Select by integer position	Explicit Syntax	Shorthand Convention
Single column from dataframe	`df.iloc[:,3]`	
List of columns from dataframe	`df.iloc[:,[3,5,6]]`	
Slice of columns from dataframe	`df.iloc[:,3:7]`	
Single row from dataframe	`df.iloc[20]`	
List of rows from dataframe	`df.iloc[[0,3,8]]`	
Slice of rows from dataframe	`df.iloc[3:5]`	`df[3:5]`
Single items from series	`s.iloc[8]`	`s[8]`
List of item from series	`s.iloc[[2,8,1]]`	`s[[2,8,1]]`
Slice of items from series	`s.iloc[5:10]`	`s[5:10]`

Figure 1.5: Dataframe Integer Indexing



pandas	Python equivalent	Meaning
`a & b`	`a and b`	`True` if both `a` and `b` are `True`, else `False`
`a | b`	`a or b`	`True` if either `a` or `b` is `True`
`~a`	`not a`	`True` if `a` is `False`, else `False`

Figure 1.6: Boolean Operators in Pandas

1.1.6 Data Cleaning Basics

- Data Encoding
- Dataframe Column Cleaning
- Dataframe Text to Numeric Cleaning
- Series.astype() method

Example 1.4: Cleaning Column Names

The column labels have a variety of upper and lowercase letters, as well as parentheses, which will make them harder to work with and read. We aim to clean our column labels by using several python string methods that will:

- *Replace spaces with underscores.*
- *Remove special characters.*
- *Make all labels lowercase.*
- *Shorten any long column names. (Specifically changing 'Operating System' to 'os')*

# CB 1.1.1 #

```
import pandas as pd
laptops = pd.read_csv('laptops.csv', encoding='Latin-1')

def mr_clean(s):
    s = s.strip()
    s = s.replace("Operating System", "os")
    s = s.replace(" ", "_")
    s = s.replace(")", "")
    s = s.replace("(","")
    sfinal = s.lower()

    return sfinal

new_columns = []
for col in laptops.columns:
    new_columns.append(mr_clean(col))

laptops.columns = new_columns
```

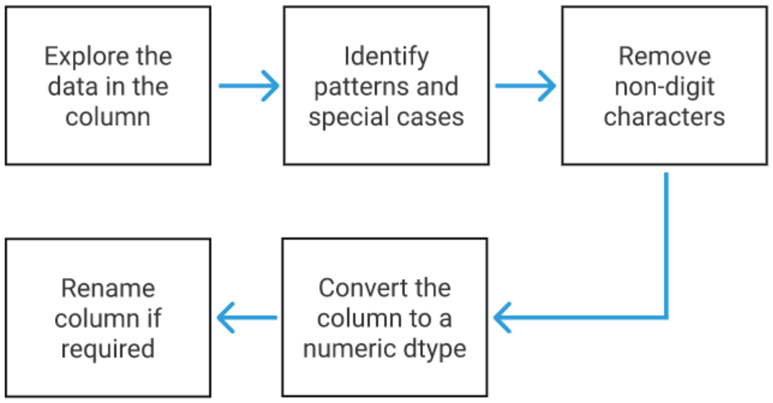

Explore the data in the column
Identify patterns and special cases
Remove non-digit characters
Convert the column to a numeric dtype
Rename column if required

Figure 1.7: Text to Numeric Data Workflow

Example 1.5: Cleaning Text into Numerical Data

We now consider the column for ram data. Exploring this column, we find a clear pattern that all stored values are strings of integers with the character 'GB' at the end of the string:

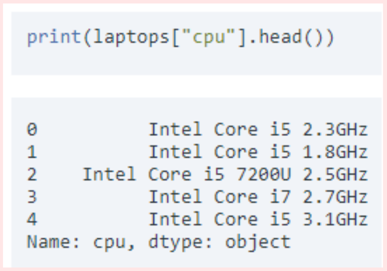
```
['8GB', '16GB', '4GB', '8GB', '32GB', '4GB', '8GB', '2GB']
```

Naturally, we want to modify the data type stored in this column so that we can easily manipulate it numerically (such as computing statistics). To do this, we'll remove the GB from the string and then recast the remaining integer-string to an integer.

```
# CB 1.1.2 #

laptops["ram"] = laptops["ram"].str.replace('GB','').astype(int)
laptops.rename({"ram":"ram_gb"},axis=1, inplace = True)
```

```
ram_gb_desc = laptops['ram_gb'].describe()
```

Example 1.6: Extracting Data from Text



```
print(laptops["cpu"].head())
```

```
0          Intel Core i5 2.3GHz
1          Intel Core i5 1.8GHz
2    Intel Core i5 7200U 2.5GHz
3          Intel Core i7 2.7GHz
4          Intel Core i5 3.1GHz
Name: cpu, dtype: object
```

Figure 1.8: First 5 entries of the Laptops[‘cpu’] column

We can observe that the CPU column in Figure 1.8 contains several pieces of information. They all appear to express the manufacturer and then model. Naturally, we want to break this apart so that we can organize it according to manufacturer and model. We use the fact spaces separate key pieces of information, allowing us to use the split() method to construct a list of strings. Selecting the first element out of this list would give us the manufacturer. Constructing a new cpu_manufacturer column is therefore achieved with the following code:

```
# CS 1.1.3 #

laptops["cpu_manufacturer"] = (laptops["cpu"]
                               .str.split()
                               .str[0]
                              )
```

The parenthesis are included so that one line of code can be expressed on multiple lines. This is done for readability purposes.

Definition 1.13: Pandas Series.map()

Series.map(self, arg, na_action=None)

Map values of Series according to input correspondence. Used for substituting each value in a Series with another value, that may be derived from a function, a dict or a Series. When arg is a dictionary, values in Series that are not in the dictionary (as keys) are converted to NaN. However, if the dictionary is a dict subclass that defines __missing__ (i.e. provides a method for default values), then this default is used rather than NaN. For further documentation, see [5].

PARAMETERS
***arg**: function, dict, or Series.*
Mapping correspondence.

***na_action**: None, ‘ignore’, default None.*
If ‘ignore’, propagate NaN values, without passing them to the mapping correspondence.

Example 1.7: Who Needs A Map?

Let s be a series containing fruit names. We note that s currently contains incorrectly spelt fruit names. Printing the series gives us a display of that incorrect spelling.

```
# CS 1.1.4 #

print(s)

Output:
0        pair
1      oranje
2     bannana
3      oranje
4      oranje
5      oranje
dtype: object
```

Hence, to fix the problem seen in CS 1.1.4, we define a dictionary called corrections that will be fed into the argument of the map() method. For each string that appears in the dictionary argument, it will be mapped to the corresponding value of that dictionary argument. For instance, pair $\mapsto$ pear.

```
# CS 1.1.5 #

corrections = {
"pair": "pear",
"oranje": "orange",
"bananna": "banana"
}
s = s.map(corrections)
print(s)

Output:
0        pear
1      orange
2      banana
3      orange
4      orange
5      orange
dtype: object
```

Handling Null Values

In pandas, null values will be indicated by either ***NaN*** *or* ***None****. There are a few main options for handling missing values:*

- *Remove any rows that have missing values.*
- *Remove any columns that have missing values.*
- *Fill the missing values with some other values.*
- *Leave the missing values as is.*

The first two options are often used to prepare data for machine learning algorithms, which are unable to *be used with data that includes null values. We can use the DataFrame.dropna() method to remove or drop rows and columns with null values.*

Example 1.8: Cleaning the Weight Column

In this example, the weights column is filled with strings of numeric values followed by a kg or kgs string that needs to be removed. After both these strings are removed, we recast the entries as floats.

```
# CS 1.1.6 #

laptops["weight"] = laptops["weight"].str.replace("kg","")
laptops["weight"] = laptops["weight"].str.replace("s","").astype(float)

laptops.rename({"weight":"weight_kg"}, axis=1, inplace = True)

laptops.to_csv('laptops_cleaned.csv', index=False)
```

A subtlety in the above approach is that once kg is removed, all entries that contained 'Xkgs' now contain 'Xs' (X denotes a numeric value), hence why I use the replace("s","") argument. In addition, I have to recast as float with the astype() method on the second line due to the weight column still containing 'Xs' entries after the first replace() call.

1.2 Exploratory Data Visualization

1.2.1 Line Charts

Definition 1.14: Data Visualization

Data visualization is the graphical representation of information and data. By using visual elements like charts, graphs, and maps, data visualization tools provide an accessible way to see and understand trends, outliers, and patterns in data.

Definition 1.15: Matplotlib Library

Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. Matplotlib can be used in Python scripts, the Python and IPython shells, the Jupyter notebook, web application servers, and four graphical user interface toolkits. The library allows us to:

- *Quickly create common plots using high-level functions.*
- *Extensively tweak plots.*
- *Create new kinds of plots from the ground up.*

Matplotlib tries to make easy things easy and hard things possible. You can generate plots, histograms, power spectra, bar charts, error charts, scatter plots, etc., with just a few lines of code.

When working with commonly used plots in matplotlib, the general workflow is:

- Create a plot using data.
- Customize the appearance of the plot.
- Display the plot.
- Edit and repeat until satisfied.

Definition 1.16: Pyplot Module

matplotlib.pyplot

Provides a MATLAB-like plotting framework.

Definition 1.17: Plot() Function

*matplotlib.pyplot.plot(*args, scalex=True, scaley=True, data=None, **kwargs)*

Plot y versus x as lines and/or markers.

PARAMETERS:
x, y*:* array-like or scalar.
The horizontal / vertical coordinates of the data points. x values are optional and default to range(len(y)). Commonly, these parameters are 1D arrays.
They can also be scalars, or two-dimensional (in that case, the columns represent separate data sets).

fmt: str, optional.
A format string, e.g. 'ro' for red circles. See the Notes section for a full description of the format strings. Format strings are just an abbreviation for quickly setting basic line properties. All of these and more can also be controlled by keyword arguments.

data*: indexable object, optional.*
An object with labelled data. If given, provide the label names to plot in x and y.

Returns:
lines - A list of Line2D objects representing the plotted data.

Example 1.9: Plotting Monthly Unemployment Trends in 1948

```
# CB 1.2.1 #

import matplotlib.pylot as plt
import pandas as pd

unrate = pd.read_csv('unrate.csv')

plt.plot(unrate['DATE'].head(12), unrate['VALUE'].head(12))
plt.xticks(rotation = 90)
plt.xlabel("Month")
plt.ylabel("Unemployment Rate")
plt.title("Monthly Unemployment Trends, 1948")
plt.show()
```

Output:

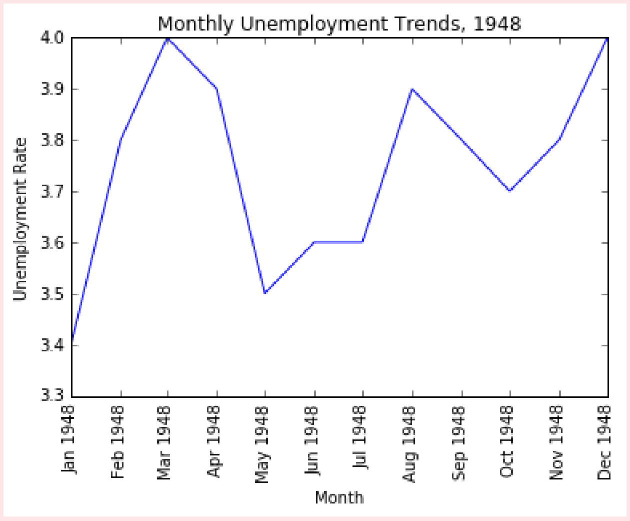


Figure 1.9: Monthly Unemployment Trends, 1948

One can notice that we didn't assign the plot in Example 1.9 to a variable and then call a method on the variable to display it. We instead called the functions plot(), show() on the pyplot module directly.

This is because every time we call a pyplot function, the module maintains and updates the plot internally (also known as state). When we call show(), the plot is displayed and the internal state is destroyed. While this workflow isn't ideal when we're writing functions that create plots on a repeated basis as part of a larger application, it's useful when exploring data.

1.2.2 Multiple Plots

When we want to work with multiple plots, however, we need to be more explicit about which plot we're making changes to. This means we need to understand the matplotlib classes that pyplot uses internally to maintain state so we can interact with them directly. Let's first start by understanding what pyplot was automatically storing under the hood when we create a single plot:

- A container for all plots was created (returned as a Figure object)
- A container for the plot was positioned on a grid (the plot returned as an Axes object)
- Visual symbols were added to the plot (using the Axes methods)

Definition 1.18: Pyplot Figure Function

*matplotlib.pyplot.figure(num=None, figsize=None, dpi=None, facecolor=None, frameon=True, clear=False, **kwargs)*

Create a new figure. For further documentation, see [6].

PARAMETERS:
***num**: integer or string, optional, default: None.*
If not provided, a new figure will be created, and the figure number will be incremented. The figure objects holds this number in a number attribute. If num is provided, and a figure with this id already exists, make it active, and returns a reference to it. If this figure does not exists, create it and returns it. If num is a string, the window title will be set to this figure's num.

figsize: *(float, float), optional, default: None.*
Width, height in inches. If not provided, defaults to rcParams["figure.figsize"] = [6.4, 4.8] = [6.4, 4.8].

dpi: *integer, optional, default: None.*
Resolution of the figure. If not provided, defaults to rcParams["figure.dpi"] = 100.0 = 100.

facecolor: *color spec.*
The background color. If not provided, defaults to rcParams["figure.facecolor"] = 'white' = 'w'.

frameon: *bool, optional, default: True.*
If False, suppress drawing the figure frame.

clear: *bool, optional, default: False*
If True and the figure already exists, then it is cleared.

Definition 1.19: Matplotlib Axes Class

*class matplotlib.axes.Axes(fig, rect, facecolor=None, frameon=True, sharex=None, sharey=None, label=", xscale=None, yscale=None, **kwargs)*

The Axes contains most of the figure elements: Axis, Tick, Line2D, Text, Polygon, etc., and sets the coordinate system.
Build an axes in a figure. For further documentation, see [7].

PARAMETERS:
fig: *Figure.*
The axes is build in the Figure fig.

rect: *[left, bottom, width, height]*
The axes is build in the rectangle rect. rect is in Figure coordinates.

sharex, sharey: *Axes, optional.*
The x or y axis is shared with the x or y axis in the input Axes.

frameon: *bool, optional.*
True means that the axes frame is visible.

Useful Matplotlib Methods

- *fig = plt.figure(figsize=(width, height))*
- *axes_obj = fig.add_subplot(nrows, ncols, plot_number)*

Example 1.10: Plotting Monthly Unemployment Trends from 1948-1952 on Subplots

```
# CB 1.2.2 #

import matplotlib.pyplot as plt
import pandas as pd
```

```
unrate = pd.read_csv('unrate.csv')

fig = plt.figure(figsize=(12,12))

axes = []
height = 5

for axnum in range(height):
    axes.append(fig.add_subplot(height,1,axnum+1))
    axes[axnum].plot(unrate['DATE'][axnum*12:(axnum+1)*12],
    unrate['VALUE'][axnum*12:(axnum+1)*12])

plt.show()
```

Output:

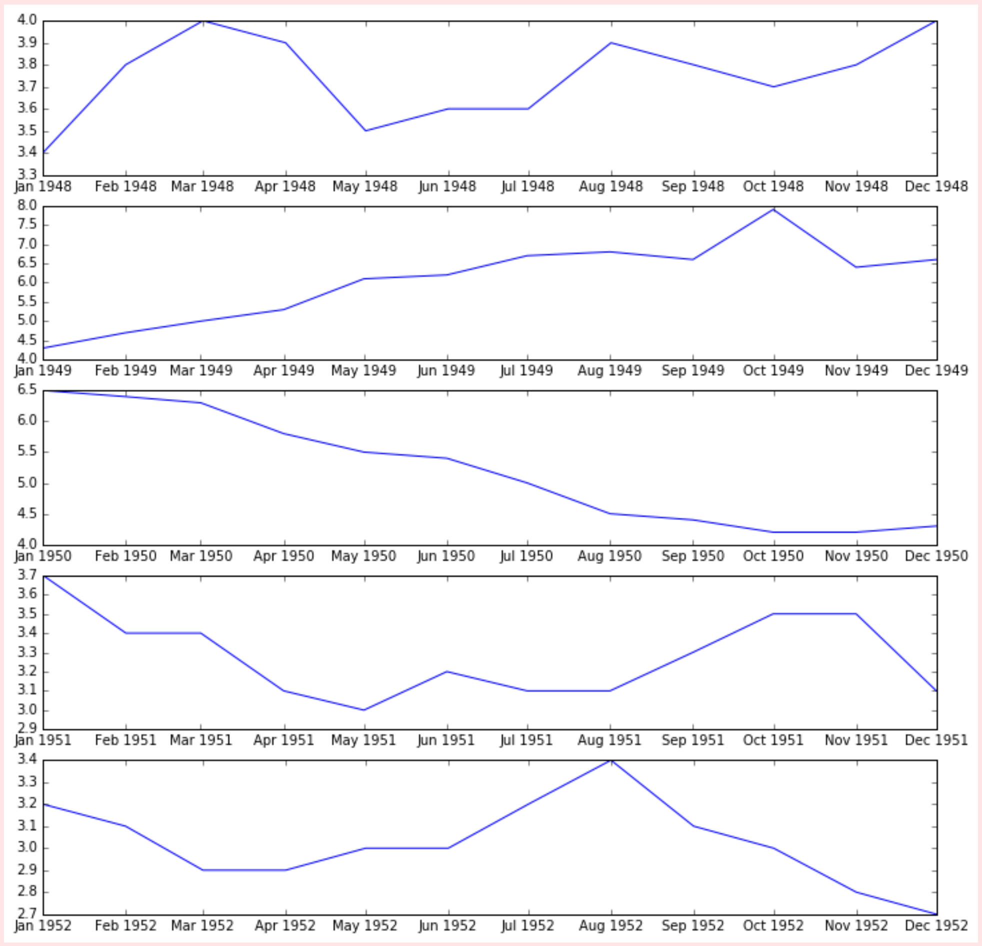


Figure 1.10: Monthly Unemployment Trends, 1948-1952 Subplots

Example 1.11: Plotting Monthly Unemployment Trends from 1948-1952 on Base Plot

```
# CB 1.2.3 #

import matplotlib.pyplot as plt
import pandas as pd

unrate = pd.read_csv('unrate.csv')

fig = plt.figure(figsize=(10,6))
colors = ['red', 'blue', 'green', 'orange', 'black']
for i in range(5):
    start_index = i*12
    end_index = (i+1)*12
    subset = unrate[start_index:end_index]
    label = str(1948 + i)
    plt.plot(subset['MONTH'], subset['VALUE'], c=colors[i], label=label)

plt.title("Monthly Unemployment Trends, 1948-1952")
plt.xlabel("Month, Integer")
plt.ylabel("Unemployment Rate, Percent")
plt.legend(loc='upper left')

plt.show()
```

Output:

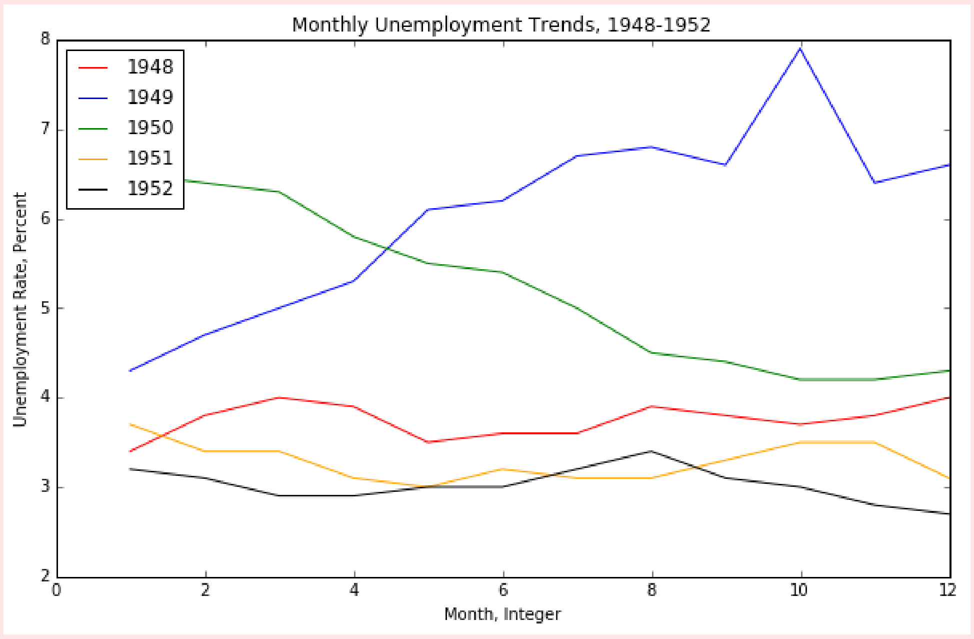


Figure 1.11: Monthly Unemployment Trends, 1948-1952

1.2.3 Bar and Scatter Plots

Example 1.12: Vertical Bar Plot for Age of Ultron Ratings

```
# CB 1.2.4 #

import matplotlib.pyplot as plt

num_cols = ['RT_user_norm', 'Metacritic_user_nom', 'IMDB_norm',
'Fandango_Ratingvalue', 'Fandango_Stars']

bar_heights = norm_reviews[num_cols].iloc[0].values
bar_positions = arange(5) + 0.75
tick_positions = range(1,6)

fig, ax = plt.subplots()
ax.bar(bar_positions, bar_heights, 0.5)
ax.set_xticks(tick_positions)
ax.set_xticklabels(num_cols, rotation=90)
ax.set_xlabel("Rating Source")
ax.set_ylabel("Average Rating")
ax.set_title("Average User Rating For Avengers: Age of Ultron (2015)")
plt.show()
```

Output:

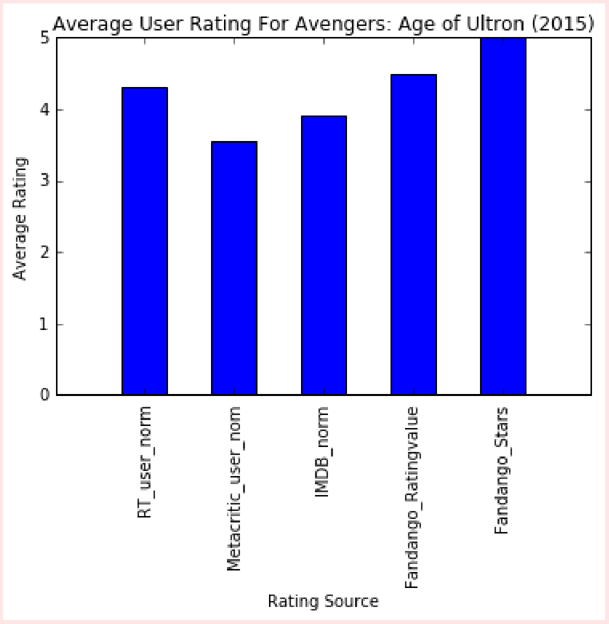


Figure 1.12: Vertical Bar Plot for Age of Ultron Ratings

Example 1.13: Horizontal Bar Plot for Age of Ultron Ratings

```
# CB 1.2.5 #

import matplotlib.pyplot as plt
from numpy import arange

num_cols = ['RT_user_norm', 'Metacritic_user_nom', 'IMDB_norm',
'Fandango_Ratingvalue', 'Fandango_Stars']

bar_widths = norm_reviews[num_cols].iloc[0].values
bar_positions = arange(5) + 0.75
tick_positions = range(1,6)

fig, ax = plt.subplots()
ax.barh(bar_positions, bar_widths, 0.5)
ax.set_yticks(tick_positions)
ax.set_yticklabels(num_cols)
ax.set_ylabel("Rating␣Source")
ax.set_xlabel("Average␣Rating")
ax.set_title("Average␣User␣Rating␣For␣Avengers:␣Age␣of␣Ultron␣(2015)")
plt.show()
```

Output:

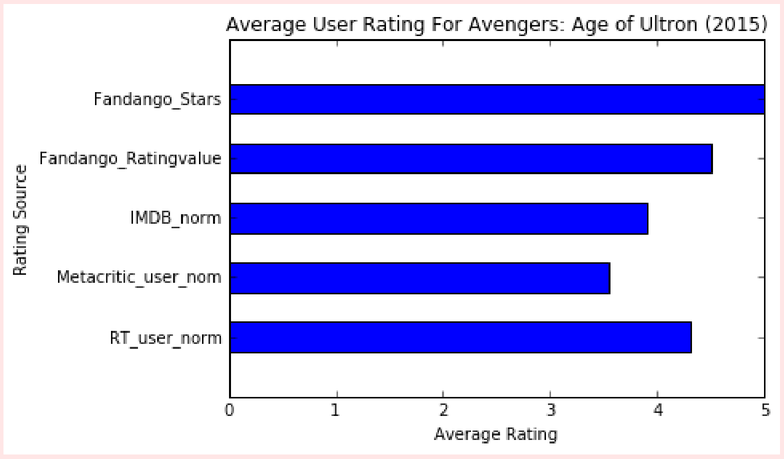


Figure 1.13: Horizontal Bar Plot for Age of Ultron Ratings

Example 1.14: Scatter Plots for Movie Rating Sites vs Fandango

```
# CB 1.2.6 #

import matplotlib.pyplot as plt

fig = plt.figure(figsize=(5,10))
ax1 = fig.add_subplot(3,1,1)
```

```
ax2 = fig.add_subplot(3,1,2)
ax3 = fig.add_subplot(3,1,3)

axes = [ax1,ax2,ax3]

review_cols = ["RT_user_norm", "Metacritic_user_norm", "IMDB_norm"]
labels = ["Rotten_Tomatoes", "Metacritic", "IMDB"]

for i in range(3):
    axes[i].scatter(norm_reviews["Fandango_Ratingvalue"],
    norm_reviews[review_cols[i]])
    axes[i].set_xlabel("Fandango")
    axes[i].set_ylabel(labels[i])
    axes[i].set_xlim(0,5)
    axes[i].set_ylim(0,5)

plt.show()
```

Output:

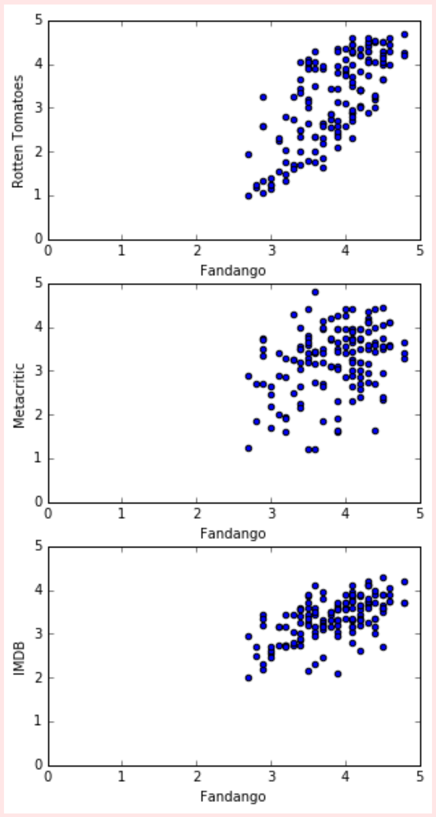


Figure 1.14: Scatter Plots among Movie Rating Sites

1.2.4 Histograms and Box Plots

Example 1.15: Movie Rating Distribution Histograms

```
# CB 1.2.7 #

import matplotlib.pyplot as plt

fig = plt.figure(figsize=(5,20))
ax1 = fig.add_subplot(4,1,1)
ax2 = fig.add_subplot(4,1,2)
ax3 = fig.add_subplot(4,1,3)
ax4 = fig.add_subplot(4,1,4)
axes = [ax1, ax2, ax3, ax4]

col_titles = ["Fandango", "Rotten Tomatoes", "Metacritic", "IMDB"]
col_names = ['Fandango_Ratingvalue', 'RT_user_norm', 'Metacritic_user_nom',
'IMDB_norm']

for i in range(4):
    axes[i].hist(norm_reviews[col_names[i]], bins =20, range = (0,5))
    axes[i].set_title("Distribution of " + col_titles[i] + " Ratings")
    axes[i].set_ylim(0,50)

plt.show()
```

Output:

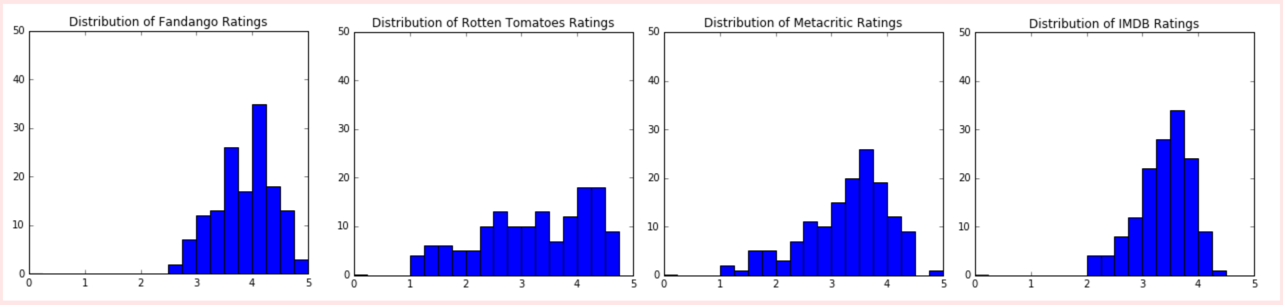


Figure 1.15: Movie Distribution Histogram Ratings

! For the sake of visual aesthetic, I've placed the plots horizontally. They are normally outputted vertically[a] in the same order presented above (Fandango on top and IMDB on bottom).

[a]In the same way as Figure 1.14.

Box Plot Quartiles

In descriptive statistics, a box plot or boxplot is a method for graphically depicting groups of numerical data through their quartiles. Box plots may also have lines extending vertically from the boxes (whiskers) indicating variability outside the upper and lower quartiles, hence the terms box-and-whisker plot and box-and-whisker diagram.

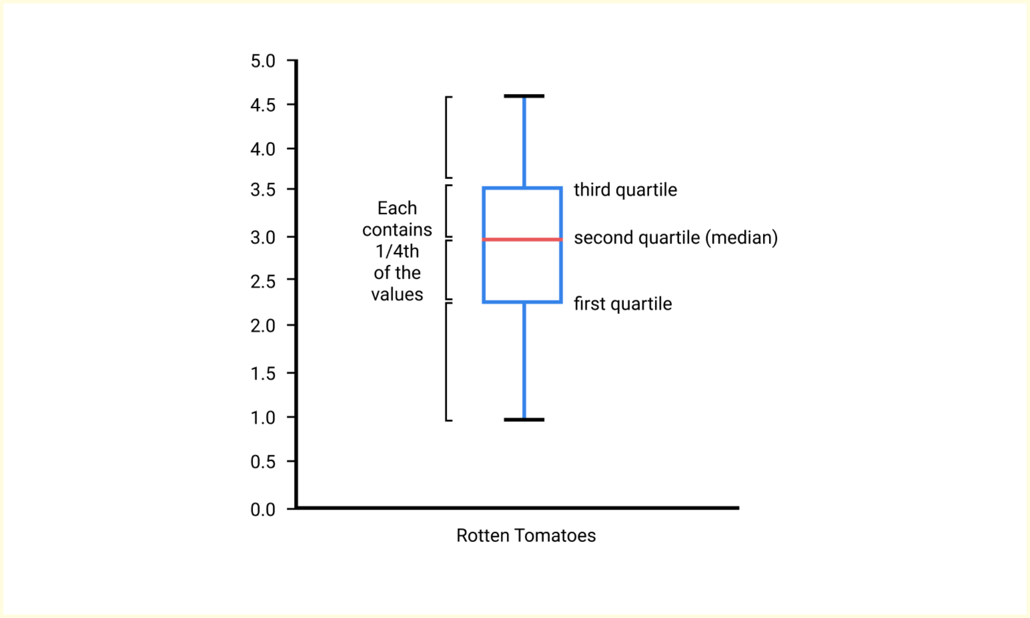


Figure 1.16: Box Plot Quartiles

We have graphically depicted an example of such a box plot in Figure 1.16 for the movie rating website Rotten Tomatoes.

Example 1.16: Movie Rating Box Plots

```
# CB 1.2.8 #

num_cols = ['RT_user_norm', 'Metacritic_user_nom', 'IMDB_norm',
'Fandango_Ratingvalue']

fix, ax = plt.subplots()

ax.boxplot(norm_reviews[num_cols].values)
ax.set_xticklabels(num_cols, rotation = 90)
ax.set_ylim(0,5)

plt.show()
```

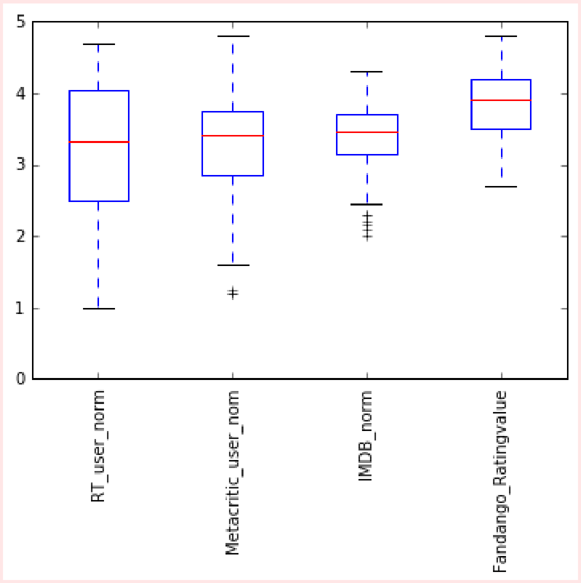


Figure 1.17: Movie Distribution Ratings Box Plot

1.2.5 Guided Project: Visualizing Majors Based on College Majors

Pandas has many methods for quickly generating common plots from data in DataFrames. Like pyplot, the plotting functionality in pandas is a wrapper for matplotlib. This means we can customize the plots when necessary by accessing the underlying Figure, Axes, and other matplotlib objects. For further documentation on visualization in Pandas, see [8].

Example 1.17: Pandas Plot on Male Dominated Undergrad Majors

```
# CB 1.2.9 #

import pandas as pd
import matplotlib.pyplot as plt

recent_grads = pd.read_csv('recent_grads.csv')
male_dominant = recent_grads[recent_grads['ShareWomen'] < 0.25]
male_dominant.plot(x = 'Major', y = 'ShareWomen',
title = 'Male-Dominated Majors', kind = 'bar', legend = None)
```

Output:

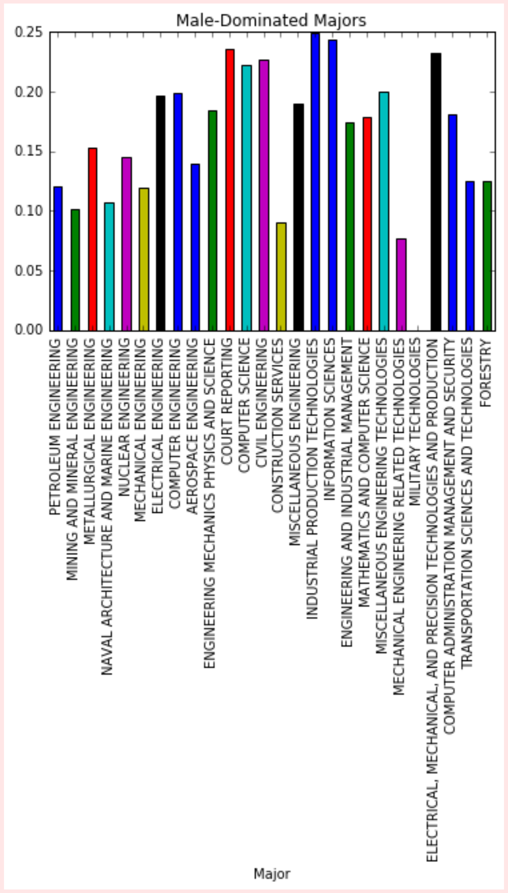


Figure 1.18: Male Dominated Majors

Example 1.18: Scatter Matrix Plots

```
# CB 1.2.10 #

import pandas as pd
import matplotlib.pyplot as plt

recent_grads = pd.read_csv('recent_grads.csv')
scatter_matrix(recent_grads[['Sample_size','Median']], figsize=(10,10))
```

Output:

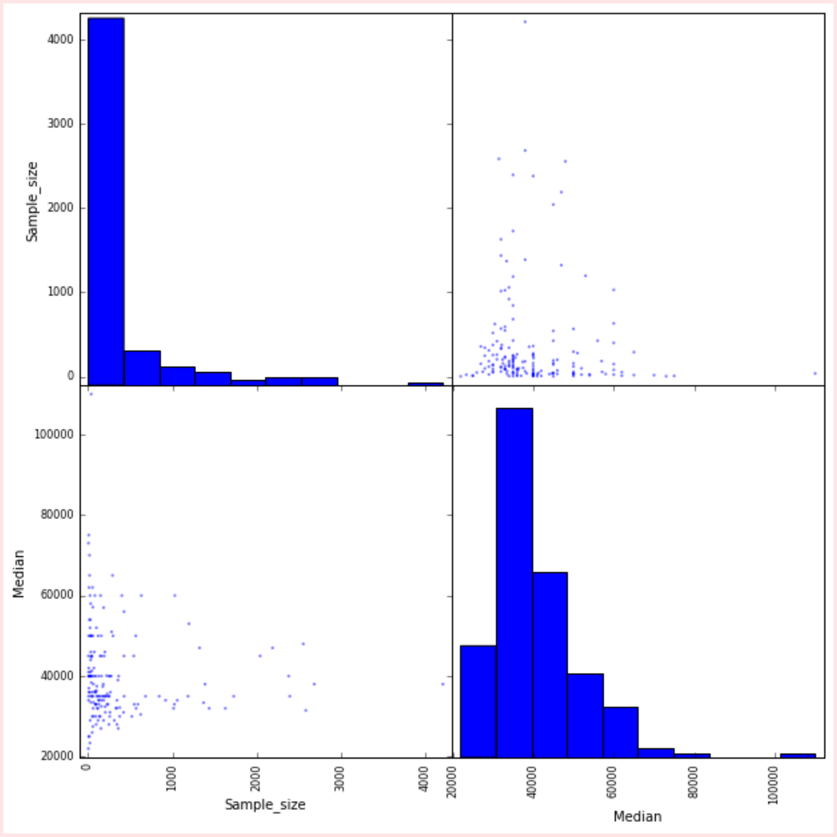


Figure 1.19: Scatter Matrix with 'Sample Size', 'Median' Columns

1.3 Storytelling Through Data Visualization

1.3.1 Improving Plot Aesthetics

Definition 1.20: Chartjunk

Chartjunk refers to all visual elements in charts and graphs that are not necessary to comprehend the information represented on the graph, or that distract the viewer from this information.

Markings and visual elements can be called chartjunk if they are not part of the minimum set of visuals necessary to communicate the information understandably. Examples of unnecessary elements that might be called chartjunk include heavy or dark grid lines, unnecessary text, inappropriately complex or gimmicky font faces, ornamented chart axes, and display frames, pictures, backgrounds or icons within data graphs, ornamental shading and unnecessary dimensions.

Example 1.19: Spine and Tick Removal

With the axis tick marks gone, the data-ink ratio is improved and the chart looks much cleaner. In addition, the spines in the chart now are no longer necessary. When we're exploring data, the spines and the ticks complement each other to help us refer back to specific data points or ranges. When a viewer is viewing our chart and trying to understand the insight we're presenting, the ticks and spines can get in the way.

```
# CB 1.3.1 #

import pandas as pd
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax.plot(women_degrees['Year'], women_degrees['Biology'], label='Women')
ax.plot(women_degrees['Year'], 100-women_degrees['Biology'], label='Men')
ax.tick_params(bottom="off", top="off", left="off", right="off")
ax.spines["right"].set_visible(False)
ax.spines["left"].set_visible(False)
ax.spines["top"].set_visible(False)
ax.spines["bottom"].set_visible(False)

ax.legend(loc='upper right')
ax.set_title('Percentage of Biology Degrees Awarded By Gender')
plt.show()
```

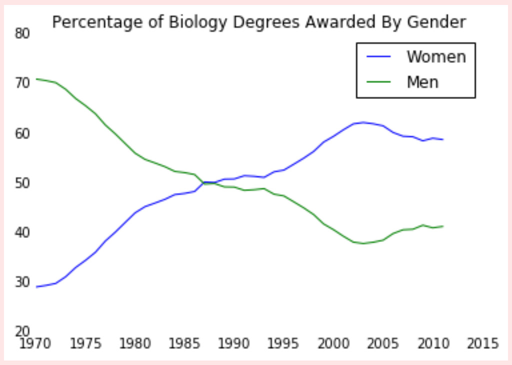


Figure 1.20: Percentage of Women Biology Degree Holders over Time

1.3.2 Conditional Plots

Definition 1.21: Seaborn Module

Seaborn is a Python data visualization library based on matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics. Seaborn works similarly to the pyplot module from matplotlib. We primarily use seaborn interactively, by calling functions in its top level namespace. Like the pyplot module from matplotlib, seaborn creates a matplotlib figure or adds to the current, existing figure each time we generate a plot.

Example 1.20: Seaborn

Under the hood, seaborn creates a histogram using matplotlib, scales the axes values, and styles it. In addition, seaborn uses a technique called kernel density estimation [9], or KDE for short, to create a smoothed line chart over the histogram.

```
# CB 1.3.2 #
```

```
import seaborn as sns
import matplotlib.pyplot as plt
sns.distplot(titanic["Age"])
plt.show()
```

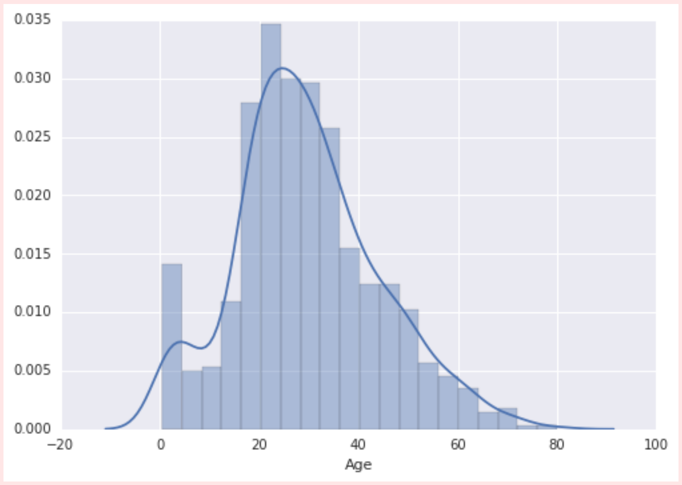


Figure 1.21: Seaborn Plot

Example 1.21: Seaborn KDE

If we wish to only view the KDE plot in seaborn, one can use the kdeplot() function. In addition, shading the area underneath the KDE function can be accomplished by setting the shade parameter to True.

```
sns.kdeplot(titanic['Age'], shade = True)
plt.xlabel("Age")
```

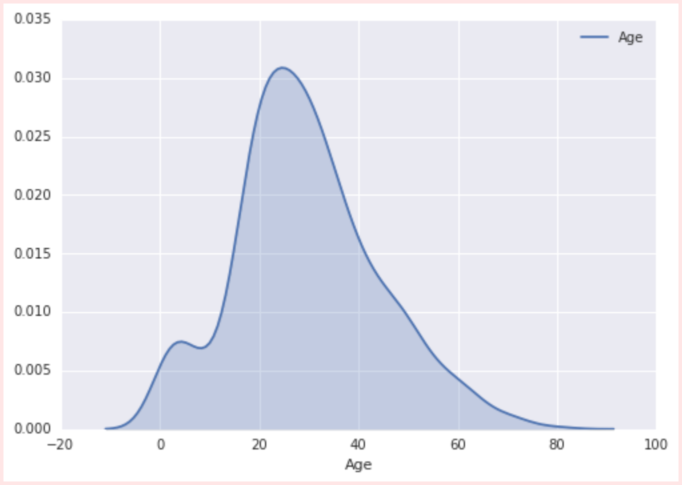


Figure 1.22: KDEplot with Shade

Example 1.22: Seaborn Despining and FacetGrid

The Seaborn despine function allows us to remove the spines from the graph. By default it will remove the top and right spines, but to remove left and bottom, we have to set these parameters to True.

The FacetGrid function allows us to display multiple graphs at the same time through conditional relationships. Along the row and column axes of the graph position in this grid, we can specify how to subset the titanic dataframe. In this example, we subset unique values for 'Pclass' along the column axis and unique values for 'Survived' along the row axis. In addition, we can also plot multiple figures on the same subplot with the hue parameter, further allowing us to subset unique values of 'Sex'.

```
# CS 1.3.3 #

g = sns.FacetGrid(titanic, col="Pclass", row="Survived", hue = 'Sex', size = 3)
g.map(sns.kdeplot, "Age", shade=True).add_legend()
sns.despine(left=True, bottom=True)
plt.show()
```

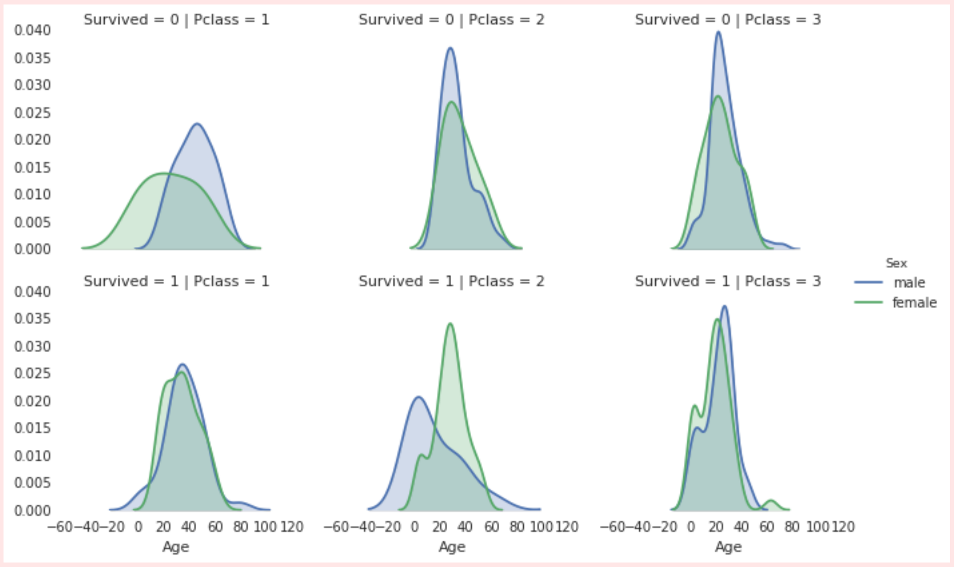


Figure 1.23: Seaborn FacetGrid with Despining

1.3.3 Visualizing Geographic Data

Definition 1.22: Basemap

```
class mpl_toolkits.basemap.Basemap
```

Sets up a basemap[10] with specified map projection. and creates the coastline data structures in map projection coordinates.

Calling a Basemap class instance with the arguments lon, lat will convert lon/lat (in degrees) to x/y map projection coordinates (in meters). The inverse transformation is done if the optional keyword inverse is set to True.

Example 1.23: Basemap Visualization with Airline Routes Data

```
# CB 1.3.4 #

import matplotlib.pyplot as plt
from mpl_toolkits.basemap import Basemap

fig, ax = plt.subplots(figsize = (15, 20))
ax.set_title('Scaled Up Earth With Coastlines')
m = Basemap(projection='merc', llcrnrlat=-80, urcrnrlat=80, llcrnrlon=-180,
urcrnrlon=180)
longitudes = airports["longitude"].tolist()
latitudes = airports["latitude"].tolist()
x, y = m(longitudes, latitudes)
m.scatter(x, y, s=1)
m.drawcoastlines()
plt.show()
```

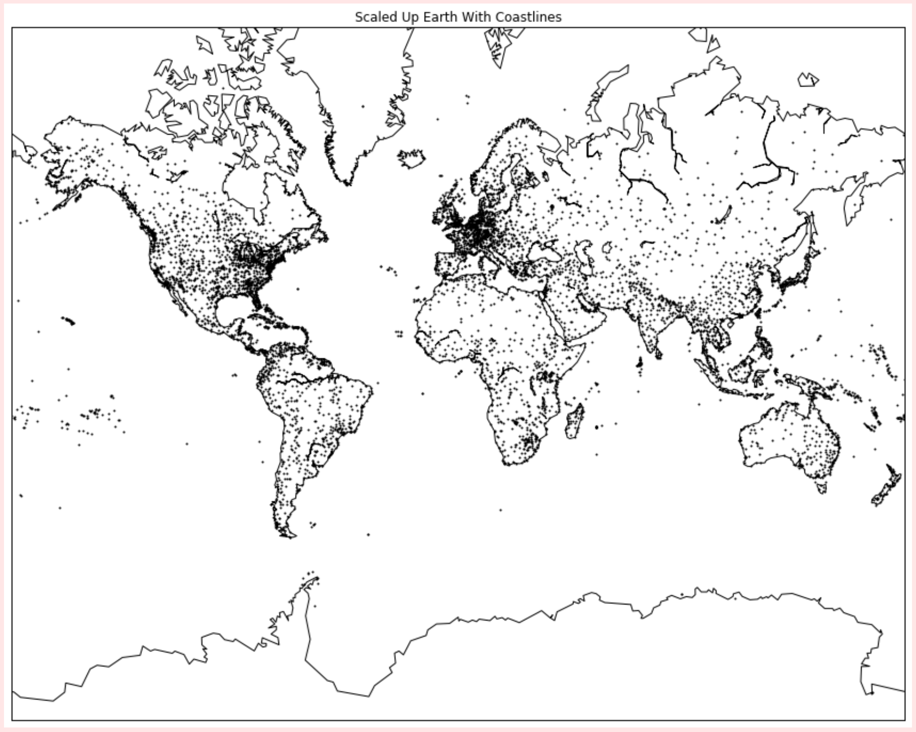


Figure 1.24: Geographic Data With Basemap

Example 1.24: Great Circles for Airline Flights

```
# CB 1.3.5 #

fig, ax = plt.subplots(figsize=(15,20))
m = Basemap(projection='merc', llcrnrlat=-80, urcrnrlat=80, llcrnrlon=-180,
 urcrnrlon=180)
m.drawcoastlines()

# Start writing your solution below this line

def create_great_circles(dataframe):

    for index, row in dataframe.iterrows():
        if abs(row['end_lat'] - row['start_lat']) < 180 and
        abs(row['end_lon'] - row['start_lon']) < 180:
            m.drawgreatcircle(row['start_lon'], row['start_lat'],
            row['end_lon'], row['end_lat'])
        else:
            continue

dfw = geo_routes[geo_routes['source']=='DFW']

create_great_circles(dfw)
plt.show()
```

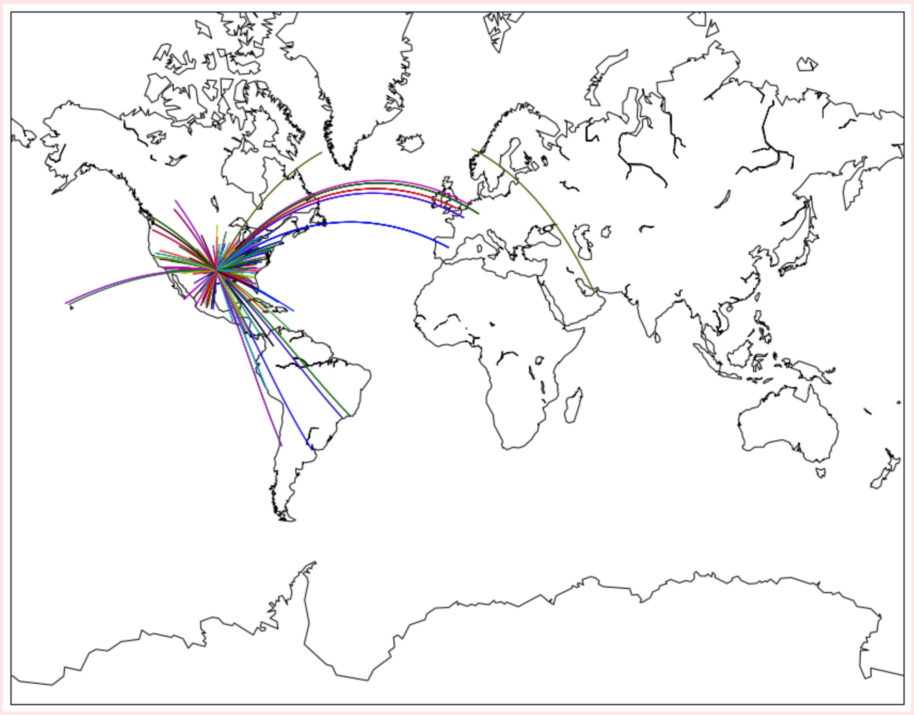


Figure 1.25: Great Circles for Airline Flights

2 The Command Line

2.1 Elements of the Command Line

2.1.1 Introduction to the Command Line

Definition 2.1: Command Line Interface

A Command Line Interface (CLI) is a text only interface through which users interact with computers by typing text instructions in a console (or terminal), using specific syntax.

As technology evolved and computers became ubiquitous, terminals were emulated within GUIs, giving rise to terminal emulators (also terminal window or just terminal).

Definition 2.2: Commands

Instructions sent to the CLI are called commands which, once input, are interpreted by a type of program called a shell or command language interpreter, and then run by your machine. Some of the most popular shells are Bash, Z shell, KornShell, Command Prompt and Windows PowerShell.

Definition 2.3: Command Parameters

The general syntax for commands are:

```
utility_name parameter1 parameter2 ... parameterN
```

By definition, a parameter is either an option *or an* argument*. An* ***option*** *is a string of symbols that modifies the behavior of the command and it always starts with a dash (-). Other possible names for this parameter are flag and switch, but depending on who you ask they might not always be interchangeable. An* ***argument*** *— or operand — is an object upon which the command acts. The* ***utility*** *(also command or program) is the first item in the instruction.*

3 Working with Data Sources

3.1 SQL Fundamentals

3.1.1 Introduction to SQL

Definition 3.1: SQL

SQL (Structured Query Language) is a domain-specific language used in programming and designed for managing data held in a relational database management system (RDBMS), or for stream processing in a relational data stream management system (RDSMS). It is particularly useful in handling structured data, i.e. data incorporating relations among entities and variables.

Definition 3.2: RDBMS

Connolly and Begg define Database Management System (DBMS) as a"software system that enables users to define, create, maintain and control access to the database". RDBMS is an extension of that acronym that is sometimes used when the underlying database is relational.

SQL Comparison Operators

We can use the following comparison operators in SQL:

- $<$: *Less than*
- $<=$: *Less than or equal to*
- $>$: *Greater than*
- $>=$: *Greater than or equal to*
- $=$: *Equal to*
- $!=$: *Not equal to*

SQL Logical Operators

We can use the following logical operators in SQL [11]:

Operator	Result
ALL	*TRUE if all of the subquery values meet the condition.*
AND	*TRUE if all the conditions separated by AND is TRUE.*
ANY	*TRUE if any of the subquery values meet the condition.*
BETWEEN	*TRUE if any of the subquery values meet the condition.*
EXISTS	*TRUE if the subquery returns one or more records.*
IN	*TRUE if the operand is equal to one of a list of expressions.*
LIKE	*TRUE if the operand matches a pattern.*
NOT	*Displays a record if the condition(s) is NOT TRUE.*
OR	*TRUE if any of the conditions separated by OR is TRUE.*
SOME	*TRUE if any of the subquery values meet the condition.*

Definition 3.3: SELECT Operation

The SELECT statement is used to select data from a database. The data returned is stored in a result table, called the result-set.

If you wish to select columns column1, column2,..., the syntax is given by:

```
SELECT column1, column2, ...
FROM table_name;
```

If you wish to select all columns from table_name, you can use ∗:

```
SELECT * FROM table_name
```

Definition 3.4: WHERE Operation

The WHERE clause is used to filter records. The WHERE clause is used to extract only those records that fulfill a specified condition.

```
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Definition 3.5: ORDER BY Operation

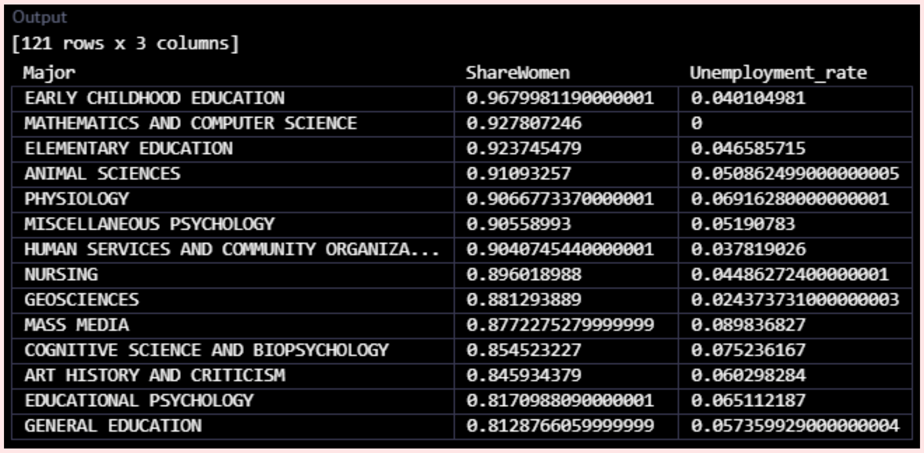
The ORDER BY keyword is used to sort the result-set in ascending or descending order.

```
SELECT column1, column2, ...
FROM table_name
ORDER BY column1, column2, ... ASC|DESC;
```

Example 3.1: Select, Where and Order

In this example, we have a table named "recent_grads", from which we want to display particular rows and columns matching our criteria. We only want to display the Major, ShareWomen and Unemployment_rate column so we use the SELECT operation to do so. We use FROM to indicate the table and WHERE to establish the two desired criterions. We want to display rows where women held at least 30% of student population and had an unemployment rate less than 10%. Finally, when we display our information, we want to order it according to the ShareWomen values, descending.

```
# CB 1.4.1 #

SELECT Major, ShareWomen, Unemployment_rate
FROM recent_grads
WHERE ShareWomen > 0.3 AND Unemployment_rate < 0.1
ORDER BY ShareWomen DESC
```



```
Output
[121 rows x 3 columns]
 Major                                      ShareWomen          Unemployment_rate
 EARLY CHILDHOOD EDUCATION                  0.9679981190000001  0.040104981
 MATHEMATICS AND COMPUTER SCIENCE           0.927807246         0
 ELEMENTARY EDUCATION                       0.923745479         0.046585715
 ANIMAL SCIENCES                            0.91093257          0.050862499000000005
 PHYSIOLOGY                                 0.9066773370000001  0.06916280000000001
 MISCELLANEOUS PSYCHOLOGY                   0.90558993          0.05190783
 HUMAN SERVICES AND COMMUNITY ORGANIZA...   0.9040745440000001  0.037819026
 NURSING                                    0.896018988         0.04486272400000001
 GEOSCIENCES                                0.881293889         0.024373731000000003
 MASS MEDIA                                 0.8772275279999999  0.089836827
 COGNITIVE SCIENCE AND BIOPSYCHOLOGY        0.854523227         0.075236167
 ART HISTORY AND CRITICISM                  0.845934379         0.060298284
 EDUCATIONAL PSYCHOLOGY                     0.8170988090000001  0.065112187
 GENERAL EDUCATION                          0.8128766059999999  0.057359929000000004
```

Figure 3.1: Displaying the result of Query CB 1.4.1

3.1.2 Summary Statistics

Definition 3.6: Aggregate Function

Aggregate functions are applied over columns of values and return a single value. MIN() and MAX(), for example, calculate and return the minimum and maximum values in a column.

Aggregate Functions [12]

Aggregate Function	*Description*
AVG	*The AVG() aggregate function calculates the average of non-NULL values in a set.*
CHECKSUM_AGG	*The CHECKSUM_AGG() function calculates a checksum value based on a group of rows.*
COUNT	*The COUNT() aggregate function returns the number of rows in a group, including rows with NULL values.*
COUNT_BIG	*The COUNT_BIG() aggregate function returns the number of rows (with BIGINT data type) in a group, including rows with NULL values.*
MAX	*The MAX() aggregate function returns the highest value (maximum) in a set of non-NULL values.*
MIN	*The MIN() aggregate function returns the lowest value (minimum) in a set of non-NULL values.*
STDEV	*The STDEV() function returns the statistical standard deviation of all values provided in the expression based on a sample of the data population.*
STDEVP	*The STDEVP() function also returns the standard deviation for all values in the provided expression, but does so based on the entire data population.*
SUM	*The SUM() aggregate function returns the summation of all non-NULL values a set.*
VAR	*The VAR() function returns the statistical variance of values in an expression based on a sample of the specified population.*
VARP	*The VARP() function returns the statistical variance of values in an expression but does so based on the entire data population.*

Definition 3.7: DISTINCT Statement

The SELECT DISTINCT statement is used to return only distinct (different) values. Inside a table, a column often contains many duplicate values; and sometimes you only want to list the different (distinct) values.

```
SELECT DISTINCT column1, column2, ...
FROM table_name;
```

Definition 3.8: SQL Alias

SQL aliases are used to give a table, or a column in a table, a temporary name. Aliases are often used to make column names more readable. An alias only exists for the duration of the query.

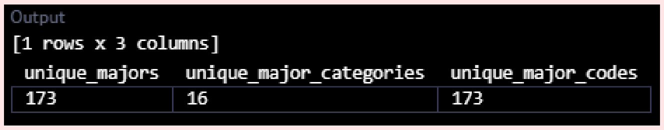
```
SELECT column_name AS alias_name
FROM table_name;
```

You may also drop the AS keyword so that the syntax can be expressed by

```
SELECT column_name alias_name
FROM table_name;
```

Example 3.2: Counting Distinct Values among Columns

```
# CB 1.4.2 #

SELECT COUNT(DISTINCT(Major)) "unique_majors",
COUNT(DISTINCT(Major_category)) "unique_major_categories",
COUNT(DISTINCT(Major_code)) "unique_major_codes"
FROM recent_grads
```

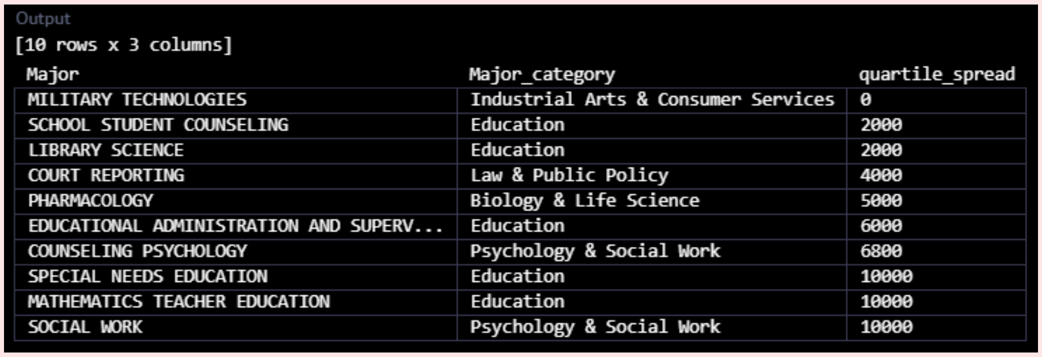


```
Output
[1 rows x 3 columns]
 unique_majors  unique_major_categories  unique_major_codes
 173            16                       173
```

Figure 3.2: The Number of Unique Values in Major, Major Category and Major Code Columns

Example 3.3: Displaying Ordered Quartile Spread among Columns

```
# CB 1.4.3 #

SELECT Major, Major_category, P75th - P25th quartile_spread FROM recent_grads
ORDER BY quartile_spread LIMIT 10
```



```
Output
[10 rows x 3 columns]
 Major                                   Major_category                      quartile_spread
 MILITARY TECHNOLOGIES                   Industrial Arts & Consumer Services  0
 SCHOOL STUDENT COUNSELING               Education                            2000
 LIBRARY SCIENCE                         Education                            2000
 COURT REPORTING                         Law & Public Policy                  4000
 PHARMACOLOGY                            Biology & Life Science               5000
 EDUCATIONAL ADMINISTRATION AND SUPERV...  Education                          6000
 COUNSELING PSYCHOLOGY                   Psychology & Social Work             6800
 SPECIAL NEEDS EDUCATION                 Education                            10000
 MATHEMATICS TEACHER EDUCATION           Education                            10000
 SOCIAL WORK                             Psychology & Social Work             10000
```

Figure 3.3: Quartile Spread for Majors

3.1.3 Group Summary Statistics

Definition 3.9: PRAGMA TABLE_INFO

The table_info pragma is used to query information about a specific table. The result set will contain one row for each column in the table. It will display information such as the type of objects stored in the columns, Null values and meanings.

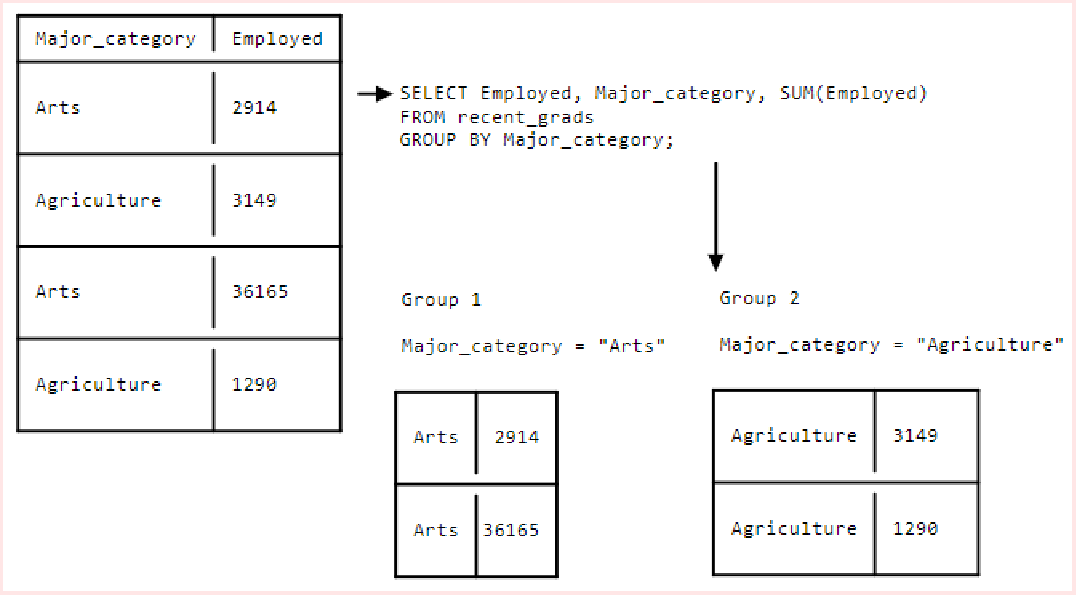
Definition 3.10: GROUP BY Statement

The GROUP BY statement groups rows that have the same values into summary rows, like "find the number of customers in each country". The GROUP BY statement is often used with aggregate functions (COUNT, MAX, MIN, SUM, AVG) to group the result-set by one or more columns.

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
ORDER BY column_name(s);
```

Example 3.4: GROUP BY in Action

The GROUP BY statement works by partitioning the relevant column into its unique entries, then performing the desired operation on each group.

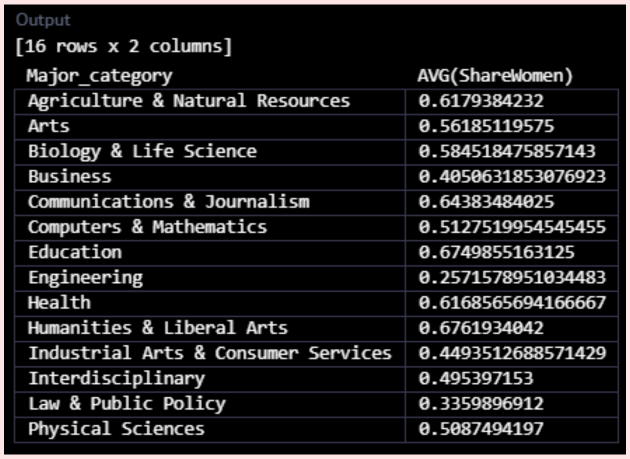


```
SELECT Employed, Major_category, SUM(Employed)
FROM recent_grads
GROUP BY Major_category;
```

Figure 3.4: Grouping Operation

In this example, we aim to compute the average of women population percentages among each major category. Once done, we display it below in Figure 3.5.

```
# CB 1.4.4 #

SELECT Major_category, AVG(ShareWomen) FROM recent_grads
GROUP BY Major_category
```



```
Output
[16 rows x 2 columns]
 Major_category                         AVG(ShareWomen)
 Agriculture & Natural Resources        0.6179384232
 Arts                                   0.56185119575
 Biology & Life Science                 0.584518475857143
 Business                               0.4050631853076923
 Communications & Journalism            0.64383484025
 Computers & Mathematics                0.5127519954545455
 Education                              0.6749855163125
 Engineering                            0.2571578951034483
 Health                                 0.6168565694166667
 Humanities & Liberal Arts              0.6761934042
 Industrial Arts & Consumer Services    0.4493512688571429
 Interdisciplinary                      0.495397153
 Law & Public Policy                    0.3359896912
 Physical Sciences                      0.5087494197
```

Figure 3.5: Group By ShareWomen Averages in Major Categories

Definition 3.11: HAVING Statement

The HAVING clause was added to SQL because the WHERE keyword could not be used with aggregate functions.

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
HAVING condition
ORDER BY column_name(s);
```

Definition 3.12: ROUND Function

The ROUND() function rounds a number to a specified number of decimal places.

```
SELECT ROUND(numeric_column_name, dec_places)
```

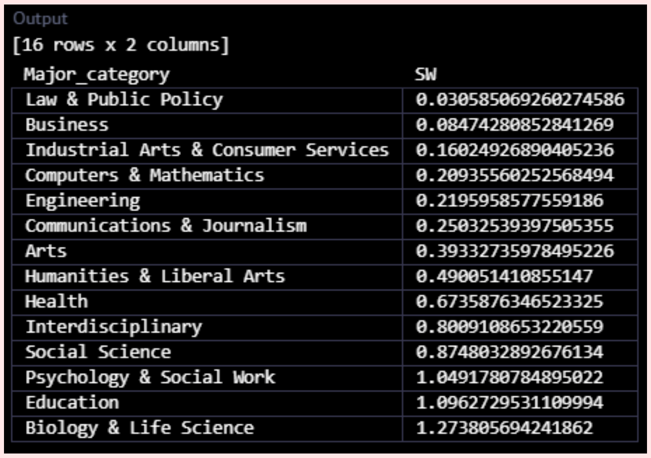
Definition 3.13: CAST Function

The CAST() function converts a value (of any type) into the specified datatype.

Example 3.5: Casting

In this example, we want to display the ratio of women to the total, grouping by each Major_category, summing and dividing by the total.

```
# CB 1.4.5 #

SELECT Major_category, CAST(Sum(Women) as float) / CAST(Sum(total) as float) SW
FROM recent_grads
GROUP BY Major_category
ORDER BY SW
```



```
Output
[16 rows x 2 columns]
 Major_category                     SW
 Law & Public Policy                0.030585069260274586
 Business                           0.08474280852841269
 Industrial Arts & Consumer Services 0.16024926890405236
 Computers & Mathematics            0.20935560252568494
 Engineering                        0.2195958577559186
 Communications & Journalism        0.25032539397505355
 Arts                               0.39332735978495226
 Humanities & Liberal Arts          0.490051410855147
 Health                             0.6735876346523325
 Interdisciplinary                  0.8009108653220559
 Social Science                     0.8748032892676134
 Psychology & Social Work           1.0491780784895022
 Education                          1.0962729531109994
 Biology & Life Science             1.273805694241862
```

Figure 3.6: The Output of Query 1.4.5

3.1.4 Subqueries

Definition 3.14: Subquery

A Subquery or Inner query or a Nested query is a query within another SQL query and embedded within the WHERE clause.A subquery is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved. Subqueries can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the operators like $=, <, >, >=, <=$ *IN, BETWEEN, etc.*

Definition 3.15: IN Operator

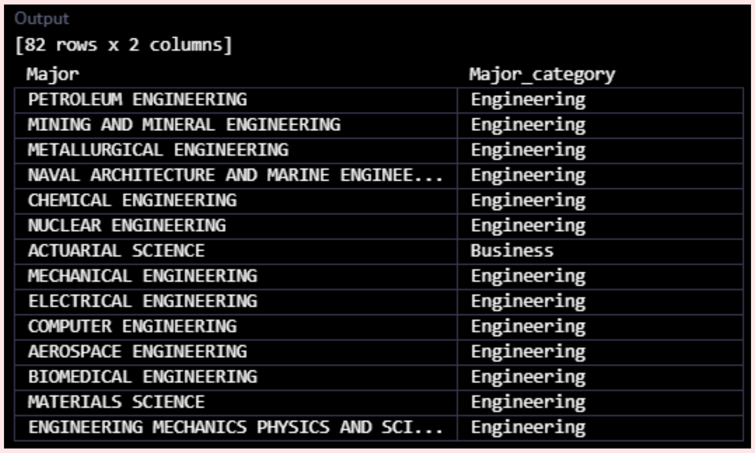
The IN operator allows you to specify multiple values in a WHERE clause. The IN operator is a shorthand for multiple OR conditions.

```
SELECT column_name(s)
FROM table_name
WHERE column_name IN (value1, value2, ...);
```

Example 3.6: Subqueries with IN Keyword

In this example, we want to write a query that returns the Major, Major_category columns for the rows where Major_category is one of the 5 highest group level sums for the Total column.

```
# CB 1.4.6 #

SELECT Major, Major_category FROM recent_grads
WHERE Major_category IN (SELECT Major_category FROM recent_grads
GROUP BY Major_category
ORDER BY SUM(Total) DESC
LIMIT 5)
```



```
Output
[82 rows x 2 columns]
```

Major	Major_category
PETROLEUM ENGINEERING	Engineering
MINING AND MINERAL ENGINEERING	Engineering
METALLURGICAL ENGINEERING	Engineering
NAVAL ARCHITECTURE AND MARINE ENGINEE...	Engineering
CHEMICAL ENGINEERING	Engineering
NUCLEAR ENGINEERING	Engineering
ACTUARIAL SCIENCE	Business
MECHANICAL ENGINEERING	Engineering
ELECTRICAL ENGINEERING	Engineering
COMPUTER ENGINEERING	Engineering
AEROSPACE ENGINEERING	Engineering
BIOMEDICAL ENGINEERING	Engineering
MATERIALS SCIENCE	Engineering
ENGINEERING MECHANICS PHYSICS AND SCI...	Engineering

Figure 3.7: The Output of Query CB 1.4.6

3.1.5 Guided Project: Analyzing CIA Factbook Data Using SQL

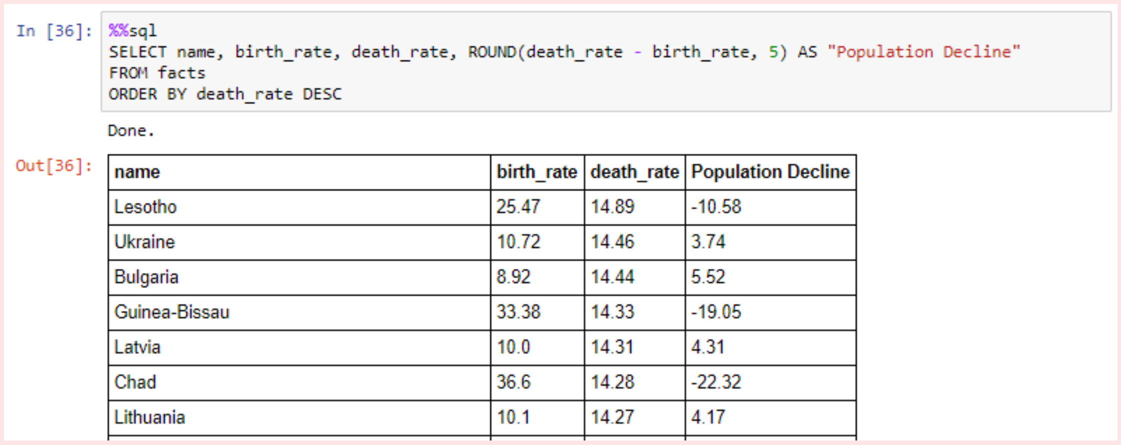
Definition 3.16: SQLite

SQLite is a relational database management system (RDBMS) contained in a C library. In contrast to many other database management systems, SQLite is not a client–server database engine. Rather, it is embedded into the end program.

Example 3.7: Analyzing CIA Factbook Data

To enable the ability to use SQL and load up the factbook.db [13], we enter the following code into the Jupyter Notebook:

```
%%capture
%load_ext sql
%sql sqlite:///factbook.db
```



```
In [36]: %%sql
SELECT name, birth_rate, death_rate, ROUND(death_rate - birth_rate, 5) AS "Population Decline"
FROM facts
ORDER BY death_rate DESC

Done.
```

Out[36]:

name	birth_rate	death_rate	Population Decline
Lesotho	25.47	14.89	-10.58
Ukraine	10.72	14.46	3.74
Bulgaria	8.92	14.44	5.52
Guinea-Bissau	33.38	14.33	-19.05
Latvia	10.0	14.31	4.31
Chad	36.6	14.28	-22.32
Lithuania	10.1	14.27	4.17

Figure 3.8: Birth / Death Rates ordered by Death Rate per Country

3.2 SQL Intermediate: Table Relations and Joins

3.2.1 Joining Data in SQL

Definition 3.17: JOIN Operation

A JOIN clause is used to combine rows from two or more tables, based on a related column between them. The syntax for a join clause is given below.

```
SELECT [column_names] FROM [table_name_one]
INNER JOIN [table_name_two] ON [join_constraint];
```

***ON**, which tells the SQL engine what columns to use to join the two tables. The syntax for specifying table column names is given by "table_name.column_name".*

Definition 3.18: Inner and Outer Join

(INNER) JOIN: *Returns records that have matching values in both tables.*
LEFT (OUTER) JOIN: *Returns all records from the left table, and the matched records from the right table.*
RIGHT (OUTER) JOIN: *Returns all records from the right table, and the matched records from the left table.*
FULL (OUTER) JOIN*: Returns all records when there is a match in either left or right table.*

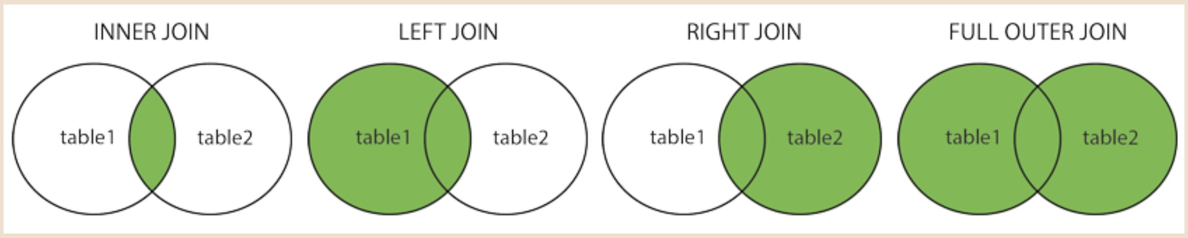
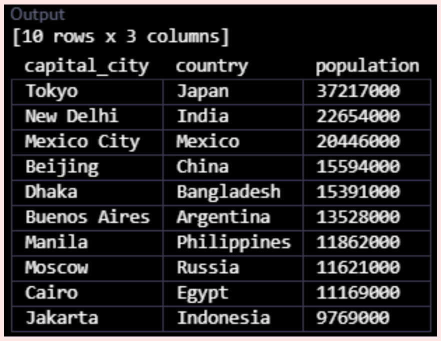


Figure 3.9: Inner and Outer Joins

Example 3.8: Population among Capital Cities

We want to write a query that returns the 10 capital cities with the highest population ranked from biggest to smallest population. It will include columns in the following order:
• *capital_city, the name of the city.*
• *country, the name of the country the city is from.*
• *population, the population of the city.*

```
# CB 1.4.7 #

SELECT c.name capital_city, f.name country, c.population population
FROM cities c
INNER JOIN facts f ON f.id = c.facts_id
WHERE c.capital = 1
ORDER BY population DESC
LIMIT 10
```



```
Output
[10 rows x 3 columns]
 capital_city  country      population
 Tokyo         Japan        37217000
 New Delhi     India        22654000
 Mexico City   Mexico       20446000
 Beijing       China        15594000
 Dhaka         Bangladesh   15391000
 Buenos Aires  Argentina    13528000
 Manila        Philippines  11862000
 Moscow        Russia       11621000
 Cairo         Egypt        11169000
 Jakarta       Indonesia    9769000
```

Figure 3.10: The Output of Query CB 1.4.7

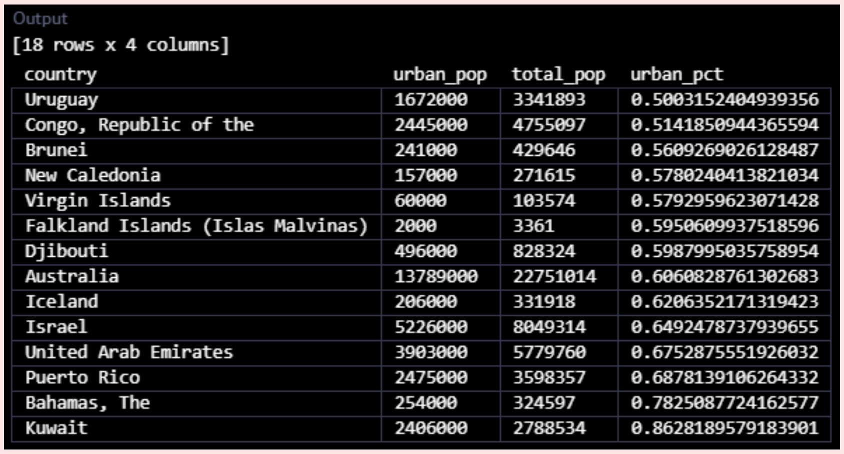
Example 3.9: Population Density in Cities Across a Country

We we will write a query that generates columns in the following order:

- *country, the name of the country.*
- *urban_pop, the sum of the population in major urban areas belonging to that country.*
- *total_pop, the total population of the country.*
- *urban_pct, the percentage of the population within urban areas, calculated by dividing urban_pop by total_pop.*

Lastly, we subject these rows to the criteria that we only want countries that have an urban_pct greater than 0.5 and that the rows should be sorted by urban_pct in ascending order.

```
# CB 1.4.8 #

SELECT f.name country, urb.urban_pop urban_pop, f.population total_pop,
CAST(urban_pop as float) / CAST(f.population as float) urban_pct
FROM facts f
INNER JOIN (SELECT SUM(c.population) urban_pop, facts_id FROM cities c
GROUP BY facts_id) urb ON urb.facts_id = f.id
WHERE urban_pct > 0.5
ORDER BY urban_pct
```



```
Output
[18 rows x 4 columns]
 country                            urban_pop  total_pop  urban_pct
 Uruguay                            1672000    3341893    0.5003152404939356
 Congo, Republic of the             2445000    4755097    0.5141850944365594
 Brunei                             241000     429646     0.5609269026128487
 New Caledonia                      157000     271615     0.5780240413821034
 Virgin Islands                     60000      103574     0.5792959623071428
 Falkland Islands (Islas Malvinas)  2000       3361       0.5950609937518596
 Djibouti                           496000     828324     0.5987995035758954
 Australia                          13789000   22751014   0.6060828761302683
 Iceland                            206000     331918     0.6206352171319423
 Israel                             5226000    8049314    0.6492478737939655
 United Arab Emirates               3903000    5779760    0.6752875551926032
 Puerto Rico                        2475000    3598357    0.6878139106264332
 Bahamas, The                       254000     324597     0.7825087724162577
 Kuwait                             2406000    2788534    0.8628189579183901
```

Figure 3.11: The Output of Query CB 1.4.8

3.2.2 Intermediate Joins in SQL

Joining more than Two Tables

The syntax for joining more than two tables in SQL is given below:

```
SELECT [column_names] FROM [table_name_one]
[join_type] JOIN [table_name_two] ON [join_constraint]
[join_type] JOIN [table_name_three] ON [join_constraint];
```

Chinook Database

In this mission, we use a database named chinook. It's schema diagram is presented below:

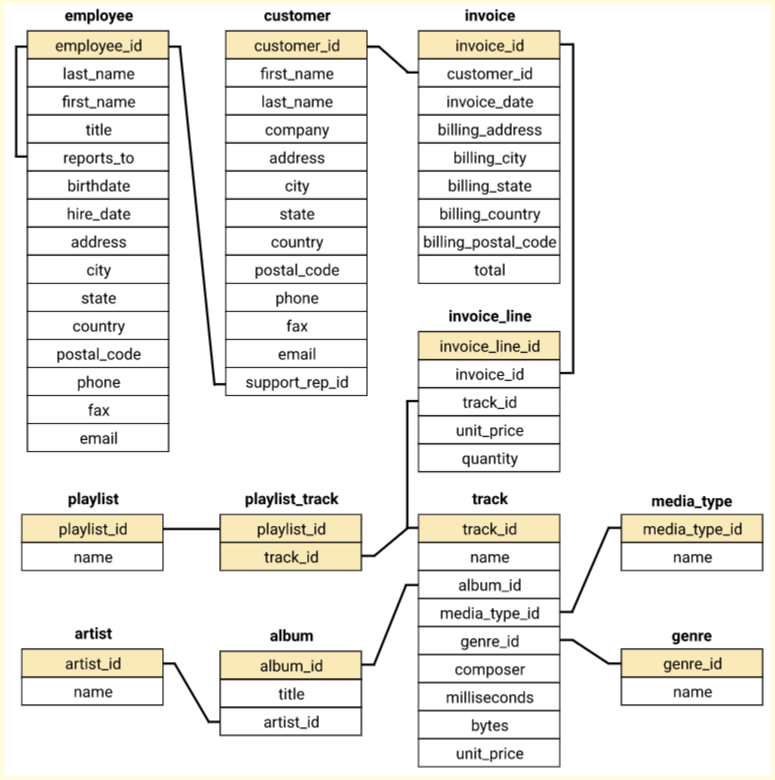


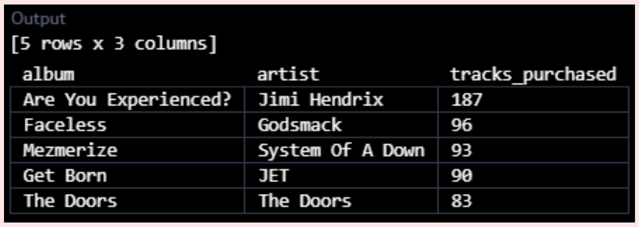
Figure 3.12: Chinook Database Schema

Example 3.10: Popular Albums

We want to write a query that returns the top 5 albums, as calculated by the number of times a track from that album has been purchased. The query will be sorted from most tracks purchased to least tracks purchased and return the following columns, in order:

- *album, the title of the album.*
- *artist, the artist who produced the album.*
- *tracks_purchased the total number of tracks purchased from that album.*

```
# CB 1.4.9 #

SELECT
    trkart.album album,
    trkart.artist_name artist,
   SUM(il.quantity) tracks_purchased
FROM invoice_line il
INNER JOIN
    (SELECT
        t.track_id,
        art.name artist_name,
            alb.title album
        FROM artist art
        INNER JOIN album alb ON alb.artist_id = art.artist_id
        INNER JOIN track t ON t.album_id = alb.album_id
      ) trkart ON trkart.track_id = il.track_id
GROUP BY album
ORDER BY tracks_purchased DESC
LIMIT 5
```



```
Output
[5 rows x 3 columns]
 album                  artist             tracks_purchased
 Are You Experienced?   Jimi Hendrix       187
 Faceless               Godsmack           96
 Mezmerize              System Of A Down   93
 Get Born               JET                90
 The Doors              The Doors          83
```

Figure 3.13: The Output of Query CB 1.4.9

Definition 3.19: Self / Recursive Join

A self JOIN is a regular join, but the table is joined with itself.

```
SELECT column_name(s)
FROM table1 T1, table1 T2
WHERE condition;
```

T1 and T2 are different table aliases for the same table.

Definition 3.20: Concatenation Operator

|| or concatenation operator is use to link columns or character strings. We can also use a literal. A literal is a character, number or date that is included in the SELECT statement.

Definition 3.21: CONCAT Function

The CONCAT() function adds two or more strings together.

```
CONCAT(string1, string2, ...., string_n)
```

Example 3.11: Employees and Supervisors

We will write a query that returns information about each employee and their supervisor. The report will include employees even if they do not report to another employee. The report will be sorted alphabetically by the employee_name column. The query will return the following columns, in order:

• ***employee_name*** *- containing the first_name and last_name columns separated by a space (eg Luke Skywalker).*
• ***employee_title*** *- the title of that employee.*
• ***supervisor_name*** *- the first and last name of the person the employee reports to, in the same format as*

employee_name.
• ***supervisor_title*** *- the title of the person the employee reports to.*

```
# CB 1.4.10 #

SELECT
    e1.first_name || "␣" || e1.last_name employee_name,
    e1.title employee_title,
    e2.first_name || "␣" || e2.last_name supervisor_name,
    e2.title supervisor_title
FROM employee e1
LEFT JOIN employee e2 ON e1.reports_to = e2.employee_id
ORDER BY employee_name
```

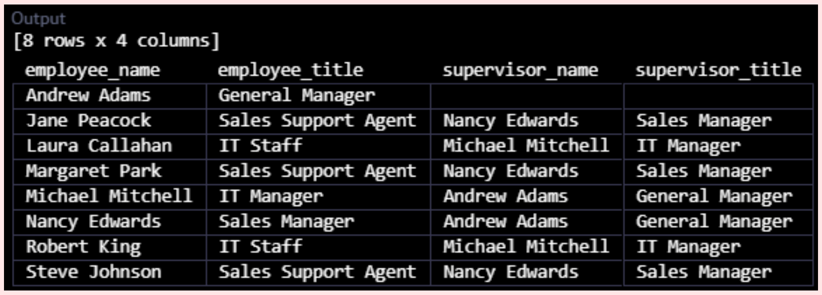

```
Output
[8 rows x 4 columns]
 employee_name     employee_title       supervisor_name    supervisor_title
 Andrew Adams      General Manager
 Jane Peacock      Sales Support Agent  Nancy Edwards      Sales Manager
 Laura Callahan    IT Staff             Michael Mitchell   IT Manager
 Margaret Park     Sales Support Agent  Nancy Edwards      Sales Manager
 Michael Mitchell  IT Manager           Andrew Adams       General Manager
 Nancy Edwards     Sales Manager        Andrew Adams       General Manager
 Robert King       IT Staff             Michael Mitchell   IT Manager
 Steve Johnson     Sales Support Agent  Nancy Edwards      Sales Manager
```

Figure 3.14: The Output of Query CB 1.4.10

Definition 3.22: LIKE Operator

The LIKE operator is used in a WHERE clause to search for a specified pattern in a column. There are two wildcards often used in conjunction with the LIKE operator:
% - The percent sign represents zero, one, or multiple characters.
_ - The underscore represents a single character.

```
SELECT column1, column2, ...
FROM table_name
WHERE columnN LIKE pattern;
```

Definition 3.23: CASE Statement

The CASE statement goes through conditions and returns a value when the first condition is met (like an IF-THEN-ELSE statement). So, once a condition is true, it will stop reading and return the result. If no conditions are true, it returns the value in the ELSE clause.

If there is no ELSE part and no conditions are true, it returns NULL.

```
CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
    WHEN conditionN THEN resultN
    ELSE result
END
```

```
AS [new_column_name]
```

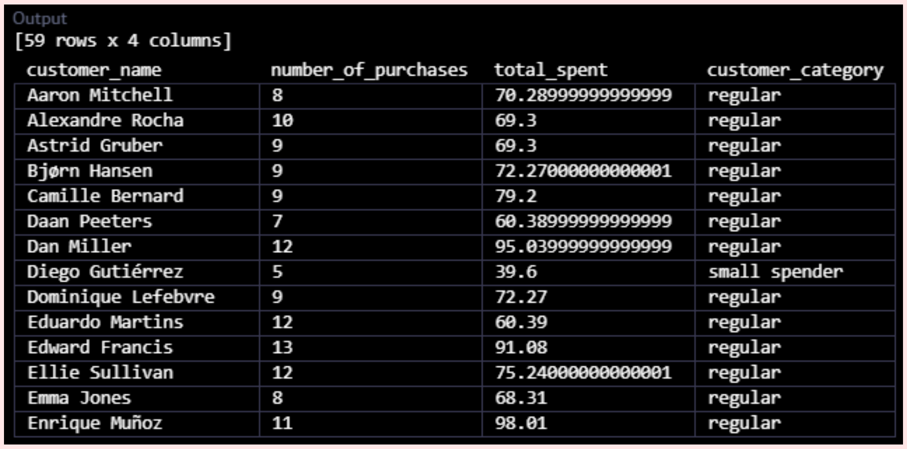
Example 3.12: Customer Spending Habits

We will write a query that summarizes the purchases of each customer. For the purposes of this exercise, we do not have any two customers with the same name. The query will include the following columns, in order:

- ***customer_name*** *- containing the first_name and last_name columns separated by a space (eg Luke Skywalker).*
- ***number_of_purchases*** *- counts the number of purchases made by each customer.*
- ***total_spent*** *- the total sum of money spent by each customer.*
- ***customer_category*** *- a column that categorizes the customer based on their total purchases. The column should contain the following values:*
    - *small spender - If the customer's total purchases are less than $40.*
    - *big spender - If the customer's total purchases are greater than $100.*
    - *regular - If the customer's total purchases are between $40 and $100 (inclusive).*

Results will be ordered by the customer_name column.

```
# CB 1.4.11 #

SELECT
      c.first_name || " " || c.last_name customer_name,
     COUNT(i.invoice_id) number_of_purchases,
     SUM(i.total) total_spent,
     CASE
          WHEN sum(i.total) < 40 THEN "small_spender"
          WHEN sum(i.total) > 100 THEN "big_spender"
          ELSE "regular"
          END
          AS customer_category
FROM customer c
LEFT JOIN invoice i ON c.customer_id = i.customer_id
GROUP BY customer_name
ORDER BY customer_name
```



```
Output
[59 rows x 4 columns]
 customer_name         number_of_purchases   total_spent         customer_category
 Aaron Mitchell        8                     70.28999999999999   regular
 Alexandre Rocha       10                    69.3                regular
 Astrid Gruber         9                     69.3                regular
 Bjørn Hansen          9                     72.27000000000001   regular
 Camille Bernard       9                     79.2                regular
 Daan Peeters          7                     60.38999999999999   regular
 Dan Miller            12                    95.03999999999999   regular
 Diego Gutiérrez       5                     39.6                small spender
 Dominique Lefebvre    9                     72.27               regular
 Eduardo Martins       12                    60.39               regular
 Edward Francis        13                    91.08               regular
 Ellie Sullivan        12                    75.24000000000001   regular
 Emma Jones            8                     68.31               regular
 Enrique Muñoz         11                    98.01               regular
```

Figure 3.15: The Output of Query CB 1.4.11

3.2.3 Building and Organizing Complex Queries

Query Readability

A little time put into whitespace and capitalization pays off. A few tips to help make your queries more readable:

- *If a select statement has more than one column, put each on a new line, indented from the select statement.*
- *Always capitalize SQL function names and keywords*
- *Put each clause of your query on a new line.*
- *Use indenting to make subqueries appear logically separate.*

Another important consideration when writing readable queries is the use of alias names and shortcuts. Name aliases should be clear– a common convention is using the first letter of the table name, however if you feel that a query is complex you should consider using more explicit aliases. Similarly, at times lines like GROUP BY 1 can be confusing, and explicitly naming the column will make your query more readable. For further information, you can also consult [14].

Definition 3.24: WITH Statement

WITH clauses allow you to define one or more named subqueries before the start of the main query. The main query then refers to the subquery by it's alias name, just as if it's a table in the database.

```
Syntax:
WITH [alias_name] AS ([subquery])

SELECT [main_query]
```

To create multiple subqueries with the WITH statement can be done by

```
WITH
    [alias_name] AS ([subquery]),
    [alias_name_2] AS ([subquery_2]),
    [alias_name_3] AS ([subquery_3])

SELECT [main_query]
```

While each subquery can be independent, we can actually use the result of the first subquery in subsequent subqueries, and so on. This can be a useful way of building readable complex queries.

Definition 3.25: CREATE VIEW

In SQL, a view is a virtual table based on the result-set of an SQL statement. A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.

You can add SQL functions, WHERE, and JOIN statements to a view and present the data as if the data were coming from one single table.

```
Syntax:
CREATE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

If you wish to add the view to a database, one can use

```
CREATE VIEW database_name.view_name AS
    SELECT * FROM database_name.table_name;
```

Definition 3.26: DROP VIEW Command

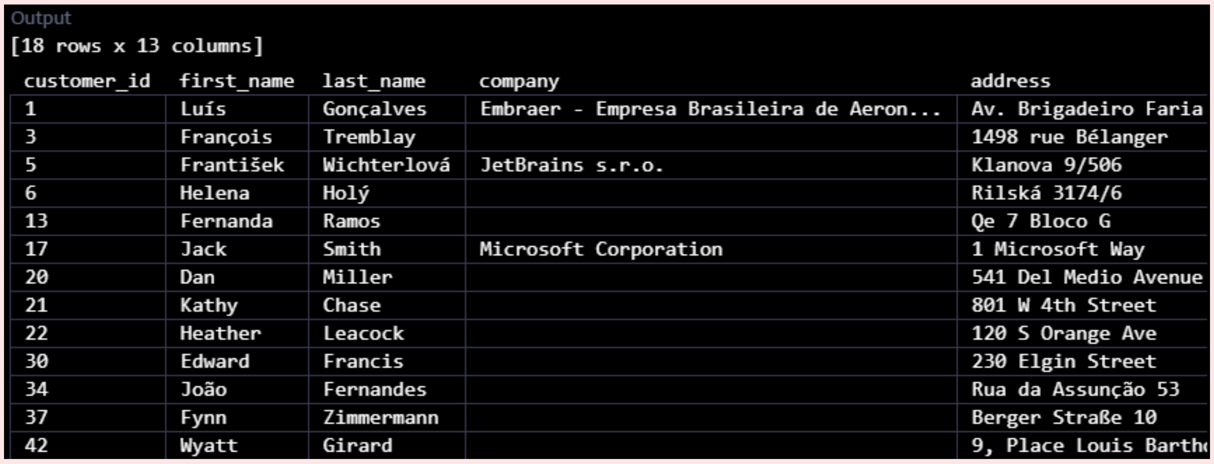
A view is deleted with the DROP VIEW command.

```
Syntax:
DROP VIEW view_name;
```

Example 3.13

We will create a view called customer_gt_90_dollars:
The view will only contain the columns from customers, in their original order.
The view will only contain customers who have purchased more than $90 in tracks from the store.
After the SQL query that creates the view, we will write a second query to display the newly created view.

```
# CB 1.4.12 #

CREATE VIEW chinook.customer_gt_90_dollars AS
    SELECT c.* FROM invoice i
    LEFT JOIN customer c ON i.customer_id = c.customer_id
    GROUP BY i.customer_id
    HAVING SUM(i.total) > 90;

SELECT * FROM chinook.customer_gt_90_dollars;
```



```
Output
[18 rows x 13 columns]
 customer_id  first_name  last_name   company                                    address
 1            Luís        Gonçalves   Embraer - Empresa Brasileira de Aeron...   Av. Brigadeiro Faria
 3            François    Tremblay                                               1498 rue Bélanger
 5            František   Wichterlová JetBrains s.r.o.                           Klanova 9/506
 6            Helena      Holý                                                   Rilská 3174/6
 13           Fernanda    Ramos                                                  Qe 7 Bloco G
 17           Jack        Smith       Microsoft Corporation                      1 Microsoft Way
 20           Dan         Miller                                                 541 Del Medio Avenue
 21           Kathy       Chase                                                  801 W 4th Street
 22           Heather     Leacock                                                120 S Orange Ave
 30           Edward      Francis                                                230 Elgin Street
 34           João        Fernandes                                              Rua da Assunção 53
 37           Fynn        Zimmermann                                             Berger Straße 10
 42           Wyatt       Girard                                                 9, Place Louis Barth
```

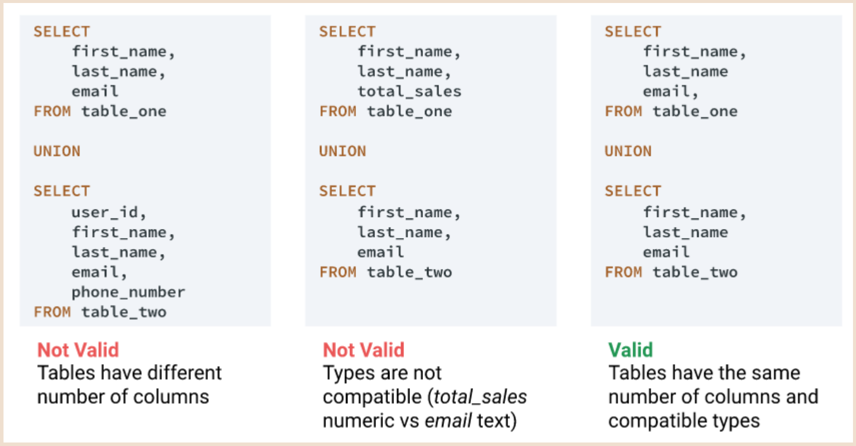
Figure 3.16: The Output of Query CB 1.4.12

Definition 3.27: UNION Operator

The UNION operator is used to combine the result-set of two or more SELECT statements.

- *Each SELECT statement within UNION must have the same number of columns.*
- *The columns must also have similar data types.*
- *The columns in each SELECT statement must also be in the same order.*

```
Syntax:
[select_statement_one]
UNION
[select_statement_two]
```



```
SELECT
    first_name,
    last_name,
    email
FROM table_one

UNION

SELECT
    user_id,
    first_name,
    last_name,
    email,
    phone_number
FROM table_two
```

Not Valid
Tables have different number of columns

```
SELECT
    first_name,
    last_name,
    total_sales
FROM table_one

UNION

SELECT
    first_name,
    last_name,
    email
FROM table_two
```

Not Valid
Types are not compatible (*total_sales* numeric vs *email* text)

```
SELECT
    first_name,
    last_name
    email,
FROM table_one

UNION

SELECT
    first_name,
    last_name
    email
FROM table_two
```

Valid
Tables have the same number of columns and compatible types

Figure 3.17: Example of SQL Union Validity

Definition 3.28: INTERSECT Operator

The SQL INTERSECT operator is used to return the results of 2 or more SELECT statements. However, it only returns the rows selected by all queries or data sets. If a record exists in one query and not in the other, it will be omitted from the INTERSECT results.

```
Syntax:
[select_statement_one]
INTERSECT
[select_statement_two]
```

Definition 3.29: EXCEPT Operator

Selects rows that occur in the first statement, but don't occur in the second statement.

```
Syntax:
[select_statement_one]
INTERSECT
[select_statement_two]
```

Example 3.14

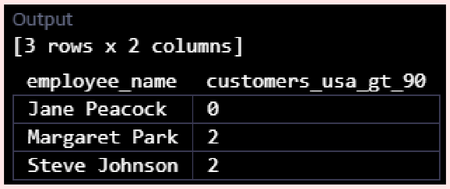
We will write a query that works out how many customers that are in the USA and have purchased more than $90 are assigned to each sales support agent. For the purposes of this exercise, no two employees have the same name.
The result will have the following columns, in order:

- *employee_name - The first_name and last_name of the employee separated by a space.*
- *customers_usa_gt_90 - The number of customer assigned to that employee that are both from the USA and have have purchased more than $90 worth of tracks.*

The result will include all employees with the title "Sales Support Agent", but not employees with any other title. Lastly, the results will be ordered by the employee_name column.

```
# CB 1.4.13 #

WITH customers_usa_gt_90 AS
    (
     SELECT * FROM customer_usa

     INTERSECT

     SELECT * FROM customer_gt_90_dollars
    )

SELECT
    e.first_name || " " || e.last_name employee_name,
    COUNT(c.customer_id) customers_usa_gt_90
FROM employee e
LEFT JOIN customers_usa_gt_90 c ON c.support_rep_id = e.employee_id
```

```
WHERE e.title = 'Sales␣Support␣Agent'
```



```
Output
[3 rows x 2 columns]
 employee_name   customers_usa_gt_90
 Jane Peacock    0
 Margaret Park   2
 Steve Johnson   2
```

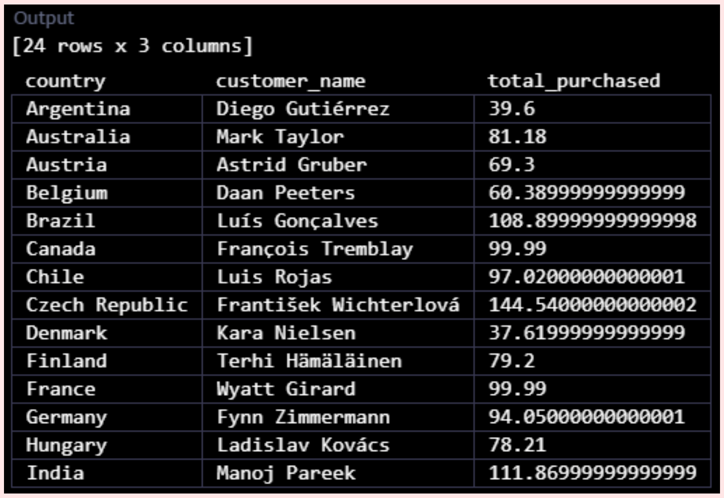
Figure 3.18: The Output of Query CB 1.4.13

Example 3.15

We will create a query to find the customer from each country that has spent the most money at the store, ordered alphabetically by country. The query will return the following columns, in order:

- *country - The name of each country that we have a customer from.*
- *customer_name - The first_name and last_name of the customer from that country with the most total purchases, separated by a space (eg Luke Skywalker).*
- *total_purchased - The total dollar amount that customer has purchased.*

```
# CB 1.4.14 #

WITH
    customer_totals AS
    (
     SELECT
         SUM(i.total) purchase_total,
          c.*
     FROM invoice i
     INNER JOIN customer c ON c.customer_id = i.customer_id
     GROUP BY c.customer_id
    ),
    customer_max AS
    (
    SELECT
        MAX(purchase_total) max_purchase,
         ct.country country,
         ct.first_name || "␣" || ct.last_name customer_name
    FROM customer_totals ct
    GROUP BY ct.country
    )

SELECT
    cm.country country,
    cm.customer_name customer_name,
    cm.max_purchase total_purchased
FROM customer_max cm
ORDER BY country
```



```
Output
[24 rows x 3 columns]
 country         customer_name        total_purchased
 Argentina       Diego Gutiérrez      39.6
 Australia       Mark Taylor          81.18
 Austria         Astrid Gruber        69.3
 Belgium         Daan Peeters         60.38999999999999
 Brazil          Luís Gonçalves       108.89999999999998
 Canada          François Tremblay    99.99
 Chile           Luis Rojas           97.02000000000001
 Czech Republic  František Wichterlová 144.54000000000002
 Denmark         Kara Nielsen         37.61999999999999
 Finland         Terhi Hämäläinen     79.2
 France          Wyatt Girard         99.99
 Germany         Fynn Zimmermann      94.05000000000001
 Hungary         Ladislav Kovács      78.21
 India           Manoj Pareek         111.86999999999999
```

Figure 3.19: The Output of Query CB 1.4.14

3.2.4 Querying SQLite from Python

Definition 3.30: SQLite Connect Function

Once we import the module, we connect to the database we want to query using the connect() function. This function requires a single parameter, which is the database we want to connect to. The connect() function returns a Connection instance, which maintains the connection to the database we want to work with.

Definition 3.31: SQLite Cursor Class

The Cursor class is a SQLite class that allows us to interact with the database such as executing SQL statements and fetching the next row of a query result set [See [15] for further information]. We will use the Cursor class to

- *Run a query against the database.*
- *Parse the results from the database.*
- *Convert the results to native Python objects.*
- *Store the results within the Cursor instance as a local variable.*

Executing Queries w/o Cursor Instance

The SQLite library actually allows us to skip creating a Cursor altogether by using the execute method within the Connection object itself. SQLite will create a Cursor instance for us under the hood and run our query against the database, allowing us to skip a step. Suppose that jobs.db is a database with a recent_grads table, then the code could look like:

```
conn = sqlite3.connect("jobs.db")
query = "select * from recent_grads;"
conn.execute(query).fetchall()
```

Definition 3.32: Cursor Fetching Methods

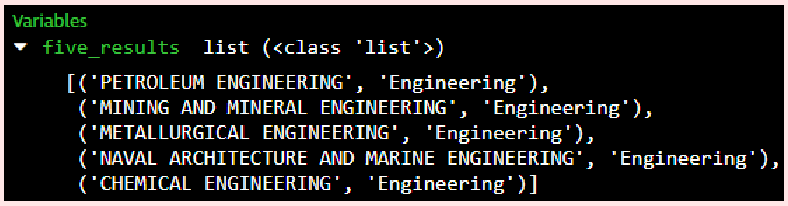
Each Cursor instance contains an internal counter that updates every time we retrieve results. When we call the fetchone() method, the Cursor instance will return a single result, and then increment its internal counter by 1. This means that if we call fetchone() again, the Cursor instance will actually return the second tuple in the results set (and increment by 1 again).

The fetchmany() method takes in an integer (n) and returns the corresponding results, starting from the current position. It then increments the Cursor instance's counter by n.

Example 3.16

In this example, we want to write and run a query that returns the Major and Major_category columns from recent_grads. We then fetch the first five results and store them as five_results.

```
# CB 1.4.15 #

import sqlite3
conn = sqlite3.connect("jobs.db")
cursor = conn.cursor()

query = "SELECT Major, Major_category FROM recent_grads"
five_results = cursor.execute(query).fetchmany(5)
```



```
Variables
▼ five_results  list (<class 'list'>)
    [('PETROLEUM ENGINEERING', 'Engineering'),
     ('MINING AND MINERAL ENGINEERING', 'Engineering'),
     ('METALLURGICAL ENGINEERING', 'Engineering'),
     ('NAVAL ARCHITECTURE AND MARINE ENGINEERING', 'Engineering'),
     ('CHEMICAL ENGINEERING', 'Engineering')]
```

Figure 3.20: The Output of CB 1.4.15

3.2.5 Guided Project: Answering Business Questions Using SQL

Definition 3.33: Context Manager

A context manager is an object that defines the runtime context to be established when executing a with statement. The context manager handles the entry into, and the exit from, the desired runtime context for the execution of the block of code. Context managers are normally invoked using the with statement (described in section The with statement), but can also be used by directly invoking their methods.

Typical uses of context managers include saving and restoring various kinds of global state, locking and unlocking resources, closing opened files, etc. [See [16] for a good article on this].

Definition 3.34: Python with statement

The with statement is used to wrap the execution of a block with methods defined by a context manager (see section With Statement Context Managers). This allows common try. . . except. . . finally usage patterns *to be encapsulated for convenient reuse.*

```
with something_that_returns_a_context_manager() as my_resource:
    do_something(my_resource)
```

The execution of the with statement with one "item" proceeds as follows:

1. The context expression (the expression given in the with_item) is evaluated to obtain a context manager.
2. The context manager's __enter__() is loaded for later use.
3. The context manager's __exit__() is loaded for later use.
4. The context manager's __enter__() method is invoked.
5. If a target was included in the with statement, the return value from __enter__() is assigned to it.

(Note: The with statement guarantees that if the __enter__() method returns without an error, then __exit__() will always be called. Thus, if an error occurs during the assignment to the target list, it will be treated the same as an error occurring within the suite would be. See step 6 below.)

6. The suite is executed.
7. The context manager's __exit__() method is invoked. If an exception caused the suite to be exited, its type, value, and traceback are passed as arguments to __exit__(). Otherwise, three None arguments are supplied.

The following code

```
with EXPRESSION as TARGET:
    SUITE
```

is semantically equivalent to:

```
manager = (EXPRESSION)
enter = type(manager).__enter__
exit = type(manager).__exit__
value = enter(manager)
hit_except = False

try:
    TARGET = value
    SUITE
except:
    hit_except = True
    if not exit(manager, *sys.exc_info()):
        raise
finally:
    if not hit_except:
        exit(manager, None, None, None)
```

Definition 3.35: Pandas read_sql_query()

pandas.read_sql_query(sql, con, index_col=None, coerce_float=True, params=None, parse_dates=None, chunksize=None

Read SQL query into a DataFrame.

Returns a DataFrame corresponding to the result set of the query string. Optionally provide an index_col parameter to use one of the columns as the index, otherwise default integer index will be used [See [17] for further information].

PARAMETERS: ***sql***: *str SQL query or SQLAlchemy Selectable (select or text object)*
SQL query to be executed.

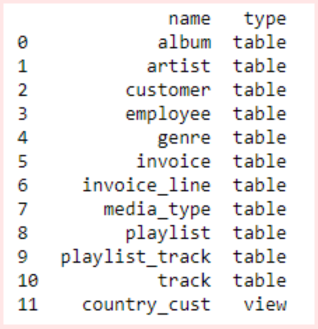
con: *SQLAlchemy connectable(engine/connection), database str URI*
or sqlite3 DBAPI2 connection. Using SQLAlchemy makes it possible to use any DB supported by that library. If a DBAPI2 object, only sqlite3 is supported.

Example 3.17: Setting up SQLite in Python

The first task is to import the SQLite, pandas and matplotlib modules, and use the magic command %matplotlib inline to make sure any plots render in the notebook.

1. Create a run_query() function, that takes a SQL query as an argument and returns a pandas dataframe of that query.
2. Create a run_command() function that takes a SQL command as an argument and executes it using the sqlite module.
3. Create a show_tables() function that calls the run_query() function to return a list of all tables and views in the database.
4. Run the show_tables() function.

```
# CB 1.4.16 #

%matplotlib inline

import sqlite3
import pandas as pd
import matplotlib.pyplot as plt

def run_query(query):
    with sqlite3.connect('chinook.db') as conn:
        return pd.read_sql_query(query, conn)

def run_command(c):
    with sqlite3.connect('chinook.db') as conn:
        conn.isolation_level = None
        conn.execute(c)

def show_tables():
    query = """SELECT name, type FROM sqlite_master
            WHERE type IN ('table', 'view') """
    tables = run_query(query)
    print(tables)

show_tables()
```



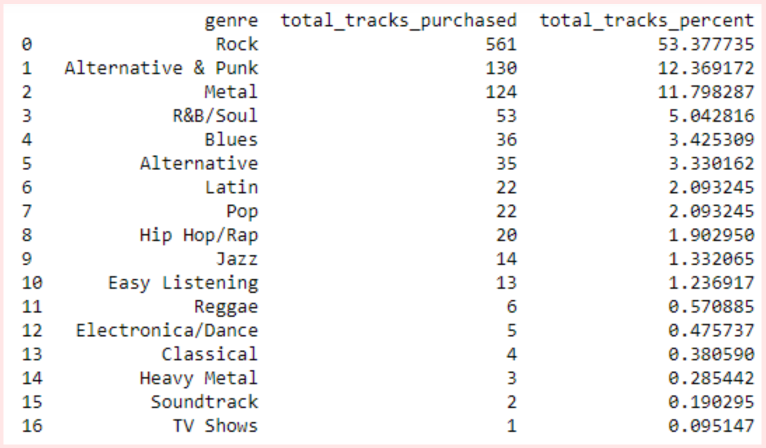
```
              name   type
0            album  table
1           artist  table
2         customer  table
3         employee  table
4            genre  table
5          invoice  table
6     invoice_line  table
7       media_type  table
8         playlist  table
9   playlist_track  table
10           track  table
11    country_cust   view
```

Figure 3.21: The Output of CB 1.4.16

Example 3.18: Finding Total Tracks Purchased in Each Genre from the Chinook Database

We want to write a query that returns each genre, with the number of tracks sold in the USA:
• in absolute numbers.
• in percentages.
We'll create a plot to show this data.

```
# CB 1.4.17 #

query_genre = """
WITH usa_tracks AS
    (
    SELECT
        i.invoice_id invoice_id,
        il.track_id track_id,
        il.quantity
    FROM invoice i
    INNER JOIN invoice_line il ON il.invoice_id = i.invoice_id
    WHERE i.billing_country = 'USA'
    ),
    usa_genre_tracks AS
    (
    SELECT
        g.name genre,
        SUM(ut.quantity) total_tracks_purchased
    FROM usa_tracks ut
    INNER JOIN track t ON t.track_id = ut.track_id
    INNER JOIN genre g ON g.genre_id = t.genre_id
    GROUP BY 1
    )

SELECT
    ugt.*,
    (
    CAST(ugt.total_tracks_purchased AS float) /
    CAST
    (
    (
    SELECT SUM(ugt.total_tracks_purchased)
    FROM usa_genre_tracks ugt
    )
    AS float
    )
    )*100 total_tracks_percent
FROM usa_genre_tracks ugt
GROUP BY 1
ORDER BY total_tracks_purchased DESC
"""

invoice_line = run_query(query_genre)
print(invoice_line)
```



```
                genre  total_tracks_purchased  total_tracks_percent
0                Rock                     561             53.377735
1   Alternative & Punk                    130             12.369172
2               Metal                     124             11.798287
3            R&B/Soul                      53              5.042816
4               Blues                      36              3.425309
5         Alternative                      35              3.330162
6               Latin                      22              2.093245
7                 Pop                      22              2.093245
8         Hip Hop/Rap                      20              1.902950
9                Jazz                      14              1.332065
10     Easy Listening                      13              1.236917
11             Reggae                       6              0.570885
12  Electronica/Dance                       5              0.475737
13          Classical                       4              0.380590
14        Heavy Metal                       3              0.285442
15         Soundtrack                       2              0.190295
16           TV Shows                       1              0.095147
```

Figure 3.22: The Output of CB 1.4.17

We can use the built-in pandas plotting methods to produce a plot for this data. We do this below.

```
# CS 1.4.18 #

invoice_line.plot(x = 'genre', y = 'total_tracks_purchased',
                  title = "Total_Number_of_Tracks_Sold_vs_Genre_(USA)",
                  kind = 'bar', legend = None)
```

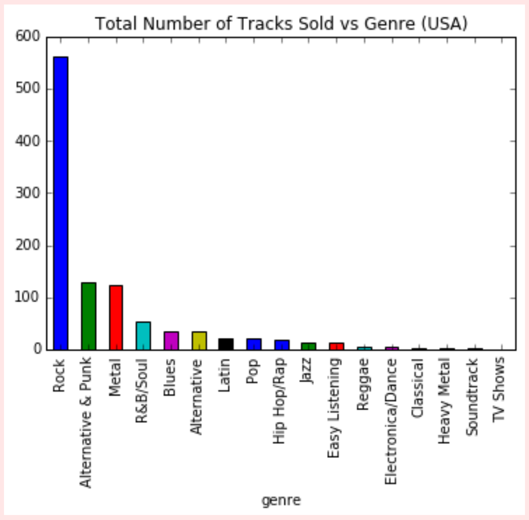


Figure 3.23: The Output of CS 1.4.18

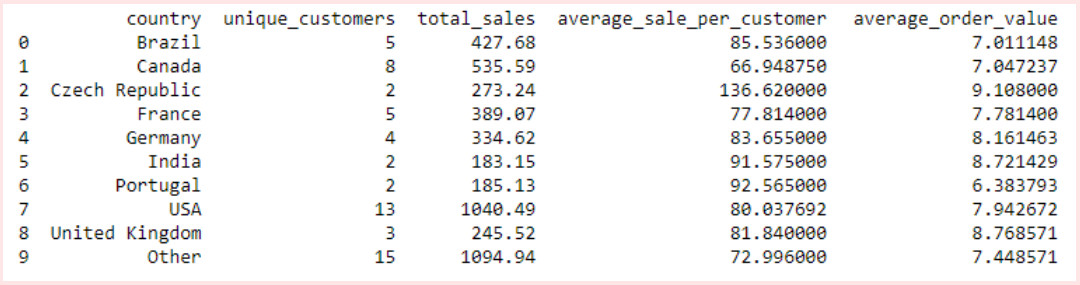
Example 3.19: Finding Transaction Details for Each Country from the Chinook Database

We will write a query that collates data on purchases from different countries. When a country has only one customer, we will collect them into an "Other" group.
The results should be sorted by the total sales from highest to lowest, with the "Other" group at the very bottom.

For each country, we will include:
- *total number of customers.*
- *total value of sales.*
- *average value of sales per customer.*
- *average order value.*

```
# CB 1.4.19 #

c = """
CREATE VIEW country_cust AS
     SELECT
          c.first_name || " " || c.last_name customer_name,
          COUNT(i.customer_id) customer_count,
          c.country country,
          SUM(i.total) total_spent
     FROM customer c
     INNER JOIN invoice i ON i.customer_id = c.customer_id
     GROUP BY 1, 1
     ORDER BY customer_count DESC
"""

d = "DROP_VIEW_country_cust"

run_command(d)
run_command(c)

query = """
With country_info AS
    (
    SELECT
        cc.country,
        SUM(cc.customer_count) orders,
        Count(cc.customer_name) unique_customers,
        Sum(total_spent) total_sales
    FROM country_cust cc
    GROUP BY cc.country
    ORDER BY total_sales
    ),
    other_info AS
    (
    SELECT
        CASE
            WHEN ci.unique_customers = 1 THEN "Other"
            ELSE 0
        END
        AS country,
        SUM(ci.orders) orders,
        Count(ci.unique_customers) unique_customers,
        SUM(ci.total_sales) total_sales
    FROM country_info ci
    WHERE ci.unique_customers = 1
    ),
    country_other_info AS
    (
    SELECT *
    FROM
        (
        SELECT
            coi.*,
            CASE
                WHEN coi.country = "Other" THEN "1"
                ELSE 0
            END
            AS count
        FROM
        (
        SELECT * FROM other_info

        UNION

        SELECT * FROM country_info ci
        WHERE ci.unique_customers != 1
        ) coi
        ORDER BY count ASC
        ) coi_new
    )
```

```
SELECT
    country,
    unique_customers,
    total_sales,
    total_sales / CAST(unique_customers as float) average_sale_per_customer,
    total_sales / CAST(orders as float) average_order_value
FROM country_other_info

"""
querydf = run_query(query)
print(querydf)
```



```
          country  unique_customers  total_sales  average_sale_per_customer  average_order_value
0          Brazil                 5       427.68                  85.536000             7.011148
1          Canada                 8       535.59                  66.948750             7.047237
2  Czech Republic                 2       273.24                 136.620000             9.108000
3          France                 5       389.07                  77.814000             7.781400
4         Germany                 4       334.62                  83.655000             8.161463
5           India                 2       183.15                  91.575000             8.721429
6        Portugal                 2       185.13                  92.565000             6.383793
7             USA                13      1040.49                  80.037692             7.942672
8  United Kingdom                 3       245.52                  81.840000             8.768571
9           Other                15      1094.94                  72.996000             7.448571
```

Figure 3.24: The Output of CB 1.4.19

The trick in placing the 'Other' column at the very end is to assign some ordered value to 'Other' with all other countries held at some other constant value. This can be seen in the country_other_info subquery in the WITH section of CB 1.4.19.

3.2.6 Table Relations and Normalization

Definition 3.36: SQLite prompt

When you launch the SQLite shell, you will be shown the SQLite prompt, seen below by sqlite>.

```
$ sqlite3 chinook.db
-- Loading resources from /home/dq/.sqliterc
SQLite version 3.21.0 2017-10-24 18:55:49
Enter ".help" for usage hints.
sqlite>
```

Definition 3.37: SQLite Dot Commands

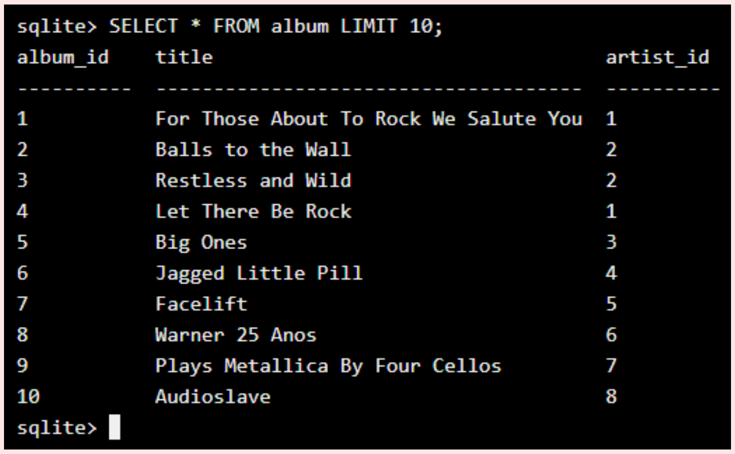
SQLite has a number of dot commands which you can use to help you work with databases. When you use a dot command, you don't need to use a semicolon. Some common dot commands are

- *.help - Displays help text showing all dot commands and their function.*
- *.tables - Displays a list of all tables and views in the current database.*
- *.shell [command] - Run a command like ls or clear in the system shell.*

- *.mode - Allows us to select from a few different display modes. We'll use .mode column to allow for easier to read outputs.*
- *.quit - Quits the SQLite shell.*

Example 3.20: Querying on SQLite Shell

We can send query instructions to the SQLite shell so that we can interact with the database.



```
sqlite> SELECT * FROM album LIMIT 10;
album_id    title                                 artist_id
----------  ------------------------------------  ----------
1           For Those About To Rock We Salute You  1
2           Balls to the Wall                      2
3           Restless and Wild                      2
4           Let There Be Rock                      1
5           Big Ones                               3
6           Jagged Little Pill                     4
7           Facelift                               5
8           Warner 25 Anos                         6
9           Plays Metallica By Four Cellos         7
10          Audioslave                             8
sqlite> 
```

Figure 3.25: SQLite Shell Query

Definition 3.38: CREATE TABLE

The CREATE TABLE statement is used to create a new table in a database.

```
CREATE TABLE [table_name] (
    [column1_name] [column1_type],
    [column2_name] [column2_type],
    [column3_name] [column3_type],
    [...]
);
```

Each column in SQLite must have a type. While some database systems have as many as 50 distinct data types, SQLite uses only 5 behind the scenes:

- *TEXT*
- *INTEGER*
- *REAL*
- *NUMERIC*
- *BLOB*

Alphabetical Index

References

[1] URL: https://docs.scipy.org/doc/numpy/reference/generated/numpy.genfromtxt.html.

[2] URL: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html.

[3] URL: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html.

[4] URL: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.html.

[5] URL: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.map.html.

[6] URL: https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.figure.html.

[7] URL: https://matplotlib.org/api/axes_api.html#matplotlib-axes.

[8] URL: https://pandas.pydata.org/pandas-docs/stable/user_guide/visualization.html.

[9] URL: https://en.wikipedia.org/wiki/Kernel_density_estimation.

[10] URL: https://matplotlib.org/basemap/api/basemap_api.html#mpl_toolkits.basemap.Basemap.

[11] URL: https://www.w3schools.com/sql/sql_operators.asp.

[12] URL: https://www.sqlservertutorial.net/sql-server-aggregate-functions/.

[13] URL: https://www.cia.gov/library/publications/the-world-factbook/.

[14] URL: https://www.sqlstyle.guide/.

[15] URL: https://docs.python.org/3/library/sqlite3.html#cursor-objects.

[16] URL: https://jeffknupp.com/blog/2016/03/07/python-with-context-managers/.

[17] URL: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_sql_query.html.