Contents

1 Introduction

1.1. Why?

Although the internet provides plenty of Python Code examples for almost all imaginable questions, these examples are not always accessible to people who are new to writing code.

Frequently, there are multiple ways of coding a solution for the same program. Access to multiple options can be confusing to newcomers as they may not have the knowledge to be able to discern which options are better than others.

Experienced coders have an understanding of the kinds of structures and algorithms they want to use, and thus know what to search for when learning a new programming language. Those who are new to coding, do not necessarily know what to search for.

1.2. Who?

Hence, this resource is for newcomers to coding with Python. There is no expectation that the reader is familiar with programming concepts. Those who are coming from another language may still find the resource helpful to identify common patterns of use.

1.3. What?

The following chapters provide an opinioned collation of exemplar snippets of code which are commonly used in data analytics. Selection of exemplars is based on (a) relevance to the data analytics topic, (b) commonly used by analysts working with Python, and (c) easy to understand by newcomers.

Satisfying these constraints may mean that the exemplars are not always the most efficient, nor the most compact code.

1.4. How?

Each chapter is arranged from easier exemplars to more complex exemplars, and so the reader is encouraged not to read from front to back, but to dip into the appropriate chapter/s to find exemplar/s that are helpful to the particular task they are working on at the time.

1.5. Where?

Exemplars are provided in 6 sections. Chapter 2 provides some basics that are used across the board and that provide a helpful foundation for more specific exemplars. Chapters 3 - 7 cover 5 topics of data analytics: (3) Reading data from external sources, (4) Cleaning data, (5) Analysing data, (6) Visualising data, and (7) Sharing data.

1.6. When?

Use the resource as you need it. It is designed to be a reference rather than a tutorial.

2

Basics

2.1. How to think

Data analytics involves 2 kinds of thinking: thinking like an analyst, and thinking like a programmer. These involve very different thought processes, and if you are aware of them it can make it easier to solve programming problems while addressing larger analytical questions.

When thinking like an analyst, questions based on the context are common. For example:

- What is the main concern of the organisation that this analysis will address?
- How should insights of this analysis be presented to the decision-makers?
- What context is important for this analysis?

The kinds of questions are high level and inform decisions about what kind of data, analysis, and visualisation is required, but may not inform decisions about how to implement the analysis.

In contrast, when thinking like a programmer, questions about the code itself are more important. For example:

- Which is the best structure for this data?
- What function is best to manipulate this data to achieve the required result?
- What syntax is required for this section of code?
- How can I break this task into smaller steps?

It is this last question in particular that newcomers find challenging. There is often an expectation to obtain a result by undertaking a simple action. But programming always involves taking tasks and breaking them up into small steps, ensuring that each step achieves the expected result. How small you have to break down the task usually depends on the degree to which others have simplified the task previously – and already written code to do the job. Although almost all data analytics with Python makes use of code written by others (to simplify the analysis task), there will always be a need to think programmatically about the code. (Or at least until AI is writing high quality code for us).

2.2. How to write

If the writing in this section didn't require any order to the words, it is likely that you wouldn't understand it. We understand a language through a knowledge of the words (the vocabulary), as well as rules for how those words are put together (syntax). The same applies to programming languages. There are certain concepts that are common across many languages (e.g. variables, loops, conditionals, data structures), but the vocabulary and the syntax tend to be language specific. It is critical to gain an understanding of both the terms important to the language *AND* the syntax required to use those terms.

In general, programming languages have very specific requirements, and python is no exception. In the following example, only the first line of code will display **Hello!** in the output. The other 2 examples will

result in errors. This is because only the first line is the *correct* syntax. Note that the words following the hash symbol (#) are comments and don't impact the running of the code in any way.

```
print('Hello!') # will work
print['Hello!'] # won't work - will produce an error
print(Hello!) # won't work - will produce an error
```

However, you might find that some things in the language don't matter. For example, with Python, it really doesn't matter if you use single quotes or double quotes for a string.

2.2.1. Nesting and indenting

One very important syntactic feature of python is the use of indents in the code to tell the computer which lines of code are *nested* inside an encapsulating element.

```
def my_function():
    print("This line of code is inside the function")

print("This line of code is outside the function")

dogs = ["Golden retriever","Australian Shepherd","Blue Heeler"] # A list of dogs
for dog in dogs: # This is a loop that loops over the list of dogs
    print("dog:",dog) # This is inside the loop

print("dogs") # This line is outside the loop
```

2.3. Common concepts

Writing code involves some common concepts no matter which language you are using. Understanding these concepts will help you grasp the language faster, and it will also help you know what to search for if you are looking for help online.

2.3.1. Variables and assignment

Variables allow us to assign data to a name. For example, I could assign the string of characters "Charles" to the variable `first_name`, and the string "Peirce" to the variable `last_name`. The advantage of doing this is that I can then operate on the variables (by name) without knowing what data is actually stored in them. For example:

```
first_name = "Charles"
last_name = "Pierce"
full_name = first_name + " " + last_name # use the variables and assign to new variable
print(full_name) # displays  Charles Pierce
```

When the string is *assigned* to the variable name, the name is *defined* as a string type (`str`), and the data (e.g. "Charles", "Pierce") is assigned to that name. When a variable is *refered* to by its name (like

in the `print()` function), the computer substitutes the data assigned to the variable name instead of the name itself. It is common to use language like *passing* a variable to a function. In the example above, `full_name` was passed to the function `print()`.

2.3.2. Functions

A **function** is simply a group of lines of code that accept data as input and return data as output. Functions are useful for repeating the same tasks repeatedly on different data. In python, the `def` keyword is used to define a function. The actual code that does the work is indented For example, the following code defines a function `make_full_name()`:

```
def make_full_name(fname,lname):
    fullname =  fname + " " + lname
    return fullname
```

This function is *called* by its name and *passing* the required parameters between the parenthesis. For example, `make_full_name("Charles","Pierce")` would result in `"Charles Pierce"`.

2.4. A little more advanced...

2.4.1. Classes with properties and methods

While functions can be created and called standalone, they can also be included in **classes**. Without going into technical details, a class can have properties and methods where the methods are functions that usually operate on properties. When classes are *instantiated* we call the instance an **object**. For example, the *object* `soccer_ball` could be an *instance* of the *class* `ball`. Using our names example above, if we had a `Person` class, it may have a `first_name` property, a `last_name` property, and a `full_name()` method. We could create :

```
# Define a class Person with properties first_name and last_name
# and a full_name method
class Person:
    def __init__(self, fname, lname):
        self.first_name = fname
        self.last_name = lname

    def full_name(self):
        return self.fname + " " + self.lname

# Create an object (instance of the Person class)
cp = Person("Charles", "Peirce")

# Get the full name for cp, by calling the full_name() method
print(cp.full_name()) # displays: Charles Peirce
```

In data analytics, there is rarely a need to create custom classes. However, it is useful to know that you may be creating and using objects and their methods. For example:

```
# Create a string object
my_string = "This is a string of characters"
# Call the string object's split() method
split_string = my_string.split('_')
print(split_string)
# Displays a  list: ['This','is','a','string','of','characters']
```

2.4.2. Libraries or Packages

Repeated code (objects and functions) that are related can be grouped together into a reusable bundle called a **Library** or **Package**. This helps minimise duplication.

In data analytics, it is very common to use *libraries* written by other programmers. Once libraries are installed, they can be *imported* into the current Python file and used by the code in that file. Libraries can be imported and used directly, imported as an alias, or components of a library can be imported. For example:

```
# Import the mathematics library
import math
# Use the library's log function by directly referring to the library
print(math.log(100)) # displays 2

# Import a function of the mathematics library
from math import sqrt
# Use the imported function
print(sqrt(100)) # displays 10

# Import a library as an alias
import pandas as pd
# Create a DataFrame object and assign to variable df
df = pd.DataFrame()
```

3
Reading

3.1. Structured data and dataframes

When working with structured data, one of the most common file types is the *CSV* (comma separated value) file. CSV files are plain text files that are structured such that every line of the file represents a row of data, and every column in the data is separated by a comma. Thus the data is structured like:

```
col1,col2,col3\n
r1c1,r1c2,r1c3\n
r2c1,r2,c2,r2c3
```

3.1.1. Reading CSV data from a file

The `pandas` library provides a function for reading CSV files directly into a dataframe. Each of the following examples requires the library, so the following `import` needs to be included prior to any of these examples.

```
import pandas as pd
```

The simplest `pandas.read_csv()` function assumes a header row and no index. After reading a CSV into a dataframe, it is a good idea to check that the shape (rows x columns) of the original file is reflected in the shape of the dataframe. The `shape` property of the dataframe will output a tuple of the total number of rows and columns (rows, cols).

```
# Read a CSV into a dataframe
file_path = "data/"
file_name = "my_data.csv"
df = pd.read_csv(f"{file_path}{file_name}")
print(df.shape) # displays (rows,cols)
```

If the first column of the data can be used as an index (each row is a unique value), then the column name (or number) can be passed.

```
# Read a CSV into a dataframe with an index on idx
file_path = "data/"
file_name = "my_data.csv"
df = pd.read_csv(f"{file_path}{file_name}",index_col='idx')
print(df.shape) # displays (rows,cols)
```

If there is no header row in the data, then one can be supplied with the `name` parameter.

```
# Read a CSV without a header into a dataframe and supply column names
file_path = "data/"
file_name = "my_data.csv"
colnames = ['col1','col2','col3']
df = pd.read_csv(f"{file_path}{file_name}",names=colnames)
print(df.shape) # displays (rows,cols)
```

3.1.2. Reading CSV data from a URL

CSV data can be read into a pandas dataframe in the same way as from a file. To check if a server supports this, you should be able to visit the URL in a web browser and see the CSV structured text.

```
# Read CSV data from a URL
url =  "https://my.data.source.api/url_endpoint"
df = pd.read_csv(url)
```

The same optional parameters (e.g. `index_col`, `names`) that are used with files will also work with a valid URL.

3.1.3. Reading Excel data

The pandas library provides additional functions for reading other structured data, including the `read_excel()` function. This facilitates opening Microsoft Excel files as dataframes directly without converting them to CSV format. Files should be in the newer `.xlsx` format.

```
# Read an Excel file
file_path = "data/"
excel_file_name = "my_excel_data.xlsx"
df = pd.read_excel(f"{file_path}{excel_file_name}")
```

If the excel file has multiple sheets, the specific sheet to read in can be specified with the `sheet_name` parameter.

```
# Read an Excel file
file_path = "data/"
excel_file_name = "my_excel_data.xlsx"
sheet = "my_sheet"
sheet_df = pd.read_excel(f"{file_path}{excel_file_name}",sheet_name=sheet)
```

3.2. Semi-structured data and JSON

3.2.1. Reading plain text files into a string

Plain text files can be read directly into a variable. After doing so, the variable contents will be a `string` of characters that are the same as the text file (including invisible characters like spaces and end of line characters).

```
# Read in a plain text file
file_path = "data/"
file_name = "my_plain_text.txt"
with open(f"{file_path}{file_name}", 'r') as file:
    text = file.read()

print(text)
```

3.2.2. Reading plain text files into a list of strings

Sometimes data saved in a text file is written in a way that each `line` of the file corresponds to a *record* in the data (e.g. CSV files, or log files). Typically an end of line character like a new line `\n` or a carriage return `\r` is used to denote the end of the line. Unix and MacOS systems tend to use `\n` while Microsoft systems often use both `\n\r`.

Python can recognise these line ending characters and read files line by line into a list. In this case, the `readlines()` function reads the file into a variable that is a `list` of `string` where each `string` is a line in the file.

```
# Read in a plain text file
file_path = "data/"
file_name = "my_plain_text.txt"
with open(f"{file_path}{file_name}", 'r') as file:
    lines_list = file.readlines()

# Display the list
print(lines_list)

# Iterate over the list and print each line of the list
for line in lines_list:
    print(line)
```

If both the full text and a list of lines is required, the data can be read using `read()` function, and then the string can be split on the end of line characters using the `split()` function.

```python
# Read in a plain text file and split into list of lines
file_path = "data/"
file_name = "my_plain_text.txt"
with open(f"{file_path}{file_name}", 'r') as file:
    full_text = file.read()

# Display the string (full_text)
print(full_text)

# Split the string into lines
lines_list = full_text.split('\n')  # <-- may need to be '\r\n'

# Dispaly the  list of strings
print(lines_list)
```

3.2.3. Reading text files in JSON format into a dict

The `JSON` format is a common file format for *semi-structured* data. The format is essentially a combination of `key:value` pairs and lists. The data is structured like:

```
{"key1":"value1",
 "key2_list": ["item1","item2","item3"],
 "key4_num": 2023,
 "key5":[45,54,63,72]}
```

This structured can be represented in Python with a combination of `dict` and `list` types.

A JSON file can be read into python by first reading the text into a `string` variable, and then converting the string into a python `dict` using the `loads()` function from the `json` library. The library needs to be imported first.

```python
import json

# Read in a plain text file
file_path = "data/"
file_json = "my_json_file.json"
with open(f"{file_path}{file_json}", 'r') as file:
    json_string = file.read()

# convert the json string
my_dict = json.loads(json_string)

# display the dict
print(my_dict)
```

If the original string is not required, the dict can be created within the `with open()` command.

```
# Read in a plain text file
file_path = "data/"
file_json = "my_json_file.json"
with open(f"{file_path}{file_json}", 'r') as file:
    my_dict = json.loads(file)

# display the dict
print(my_dict)
```

TIP: See section **??** for exemplars on how to access the data within a dict.

3.3. Other factors when reading files

3.3.1. Reading alternative encodings

When reading data from a text file (e.g. CSV), at times the file may be encoded differently than the 'utf-8' encoding that Python expects. In these cases, an alternative coding will need to be specified.

Common alternative encodings include `iso-8859-1`, and an extension to this standard by Microsoft: `cp1252`.

```
# Read a CSV into a dataframe with alternative encoding
file_path = "data/"
file_name = "my_data.csv"
df = pd.read_csv(f"{file_path}{file_name}", encoding='cp1252')

# Read a text file as JSON with alternative encoding
with open(f"{file_path}{file_json}", 'r', encoding='iso-8859-1') as file:
    my_dict = json.loads(file)
```

4

Cleaning

Data are rarely without problems. Aspects of the data can be missing, or incomplete, or even false. Data analytics can involve a significant amount of time dealing with inconsistencies and anomalies in the original data prior to starting the main analysis. This process of transforming *dirty* data into something usable is known as **cleaning**. The following examples show some common code that is useful in the cleaning process.

4.1. Not a Number (NaN)

When a dataframe expects a value to be present, but it is missing (*null*), a `NaN` fills the cell in the place of the expected value. A helpful cleaning process is to identify which columns include NaNs and replace them with a meaningful value.

Important: a meaningful replacement value is dependent on the meaning of the data in the column. For example, if a column includes measured temperatures from a sensor, a `NaN` may mean that the sensor is offline. In this case, replacing the value with a `0` would impact the truth of the data (unless the temperature was actually 0 at the time).

To see the number of NaNs in each column in the dataframe the `isna()` function can be used together with `sum()`.

```
# display the number of NaNs in each column of the data
print(df.isna().sum())
```

To replace all NaNs in a particular column with a given value, use `fillna()`.

```
# replace NaNs in col1 with string 'OTHER'
column = 'col1'
replace_value = 'OTHER'
df[column] = df[column].fillna(replace_value)
```

The same code will work for replacing multiple columns. However, take care that the replacement data means the same thing for each of the columns.

```
# replace NaNs in multiple columns with the string 'N/A'
columns = ['col1','col5']
replace_value = 'N/A'
df[columns] = df[columns].fillna(replace_value)
```

4.2. Whitespace

Sometimes cells look empty, but they actually contain invisible characters (like a space: `' '`). To see whether a column includes *whitespace*, turn the column into a list and print the list. Note, this may not be a good idea if you have a lot of rows in your dataframe!

```
# column as a list
column_name = 'col1'
print(list(df[column_name]))
```

To replace all whitespace, use the `replace()` function of the dataframe. By using **Regex** (regular expressions), it is possible to replace empty strings as well as strings with whitespace characters. The regex pattern is `r'^\s*$'` and it requires that the python regex library has been imported.

```
# import the regex library
import re
# Replace whitespace major column
column_name = 'col1'
replace_value = '_empty_'
df[column_name] = df[column_name].replace(r'^\s*$', replace_value, regex=True)
```

4.3. Converting Types

Sometimes data appears correct, but it is actually stored incorrectly in the dataframe. For example, a numeric value stored as a string or a date stored as a string. In these cases, it is necessary to convert the column to the correct type.

To identify the types of the columns in a dataframe, call the `dtypes` property.

```
df.dtypes
```

4.3.1. Converting strings to datetime

```
# If a date is a string in the format 31-12-2020, use %d-%m-%Y as the formatter
df["Date"] = pd.to_datetime(df["Date"], format=%d-%m-%Y)
```

A list of date string formatters can be found here: `https://pandas.pydata.org/docs/reference/api/pandas.Period.strftime.html`.

4.3.2. Converting strings to numeric

```
# Replace a specific non-numeric character like a comma
df["Converted"] = pd.to_numeric(df['str_number'].str.replace(',', ''))

# Using regex, we can replace all non-numeric characters and import the column as numeric
import re
df["Converted"] = pd.to_numeric(df['str_number'].str.replace('[^0-9.]', '',regex=True))

# If the only non-numeric character is a thousands separator, we can deal with it on import
df = pd.read_csv("my_file.csv",thousands=",")
```

5 Analysing

5.1. Accessing data in a dataframe

```
import pandas as pd
data = {"pet":['dog','cat','cat','dog','dog','dog'],
"colour":["black","brown","black","brown","black","brown"],
"count":[3,1,4,2,1,2]}
df = pd.DataFrame(data)
df
```

	pet	colour	count
0	dog	black	3
1	cat	brown	1
2	cat	black	4
3	dog	brown	2
4	dog	black	1
5	dog	brown	2

Get the number of data rows in the dataframe:

```
df.index.size
# displays: 6
```

Get the number of data rows and columns (not including the index) as a tuple:

```
df.shape
# displays: (6,3)
```

Get the columns of a dataframe:

```
df.columns
```

Get the index of a dataframe:

```
df.index
```

Get a column of a dataframe as a *Series*:

```
df['pet']
```

Get a row of a dataframe by its index:

```
df.loc[3]
# or if the index is a unique string or date
df.loc['unique_id']
```

Get a single value from a dataframe by its row, column label:

```
df.at[3,'pet']
# displays: 'dog'
```

5.1.1. Filtering by value

String value contains:

```
df[df['pet'].str.contains('dog')]
# returns just rows where Pet column includes Dog
```

Multiple filters (note the or `|` operator joining each truth condition):

```
df[df['pet'].str.contains('dog') | df['pet'].str.contains('cat')]
# returns all rows where Pet column includes Dog OR Cat
```

Numerical comparisons that result in a boolean value:

```
df[df['count']<=2]
# returns all rows where Count column is less than or equal to 2
```

5.2. Quantitative data

5.2.1. Descriptive Statistics

Obtain summary descriptive statistics for a dataframe. This returns a dataframe with only columns that have quantitative data, and the rows `count, mean, std, min, 25\%, 50\%, 75\%, max`:

```
df.describe()
```

Access a statistical value of describe for a selected column:

```
df['selected_column'].describe()['mean'] # displays the value of the mean for selected_column
```

5.2.2. Correlation

Create a corrlation matrix between selected quantitative columns:

```
df['col1','col2','col3'].corr()
# displays a 3x3 matrix with index matching column headings
# containing correlation values
```

5.3. Counting categorical data

Count the number of ‘dog’ and ‘cat’ entries in the ‘pet’ column:

```
result = df.groupby(['pet']).size()
print(result)
```

```
pet
cat    2
dog    4
dtype: int64
```

Get a total number of dogs and cats by using `sum()` on the ‘count’ column:

```
result = df.groupby(['pet'])['count'].sum()
print(result)
```

```
pet
cat    5
dog    8
Name: count, dtype: int64
```

Get totals based on categories and subcategories:

```
result = df.groupby(['pet','colour']).sum()
print(result)
```

```
pet colour  count
cat black       4
    brown       1
dog black       4
    brown       4
```

Get a total number of coloured pets and format them as a dataframe:

```
new_df = df.groupby(['colour'])['count'].sum().reset_index(name="count")
new_df
```

	colour	count
0	black	8
1	brown	5

5.4. Accessing data within a dicts and lists

When semi-structured data saved in `JSON` format is read into a Python, a combination of `dict` and `list` types is used to store the data. When analysing data in this format, it is necessary to *traverse* the structure to access nested data. This involves a combination of (1) accessing values via their keys, and (2) iterating over lists.

5.4.1. Accessing values via their keys

It is common for json data to be nested. For example, in the data below, the person's birth year is found in a dict with a key 'dates' inside another dict with a key 'details' which is inside another dict.

```
{
    "person_id": 1,
    "name": "Charles Peirce",
    "details": {
        "occupation":"Philosopher",
        "dates": {
            "born": 1839
            "died": 1914
        }
    }
}
```

Data in a dict is accessed via its `key`. If the data above was loaded into a variable called `peirce`, the `person_id` value could be retrieved as follows:

```
# A dict variable called peirce for the peirce record
peirce = {
    "person_id": 1,
    "name": "Charles Peirce",
    "details": {
        "occupation":"Philosopher",
        "dates": {
            "born": 1839
            "died": 1914
        }
    }
}

# Get the person_id for peirce record
peirce_id = peirce['person_id']

# Get the name for the peirce record
```

To access data deeper in the structure, the keys can be chained. . .

```
# Chain keys to get nested data
peirce_occupation = peirce['details']['occupation']

# Go as deep as the structure allows
peirce_born = peirce['details']['dates']['born']
```

Dicts can be very large, and often difficult to read. It can be helpful to identify what the keys are at various levels of the dict. This can be done using the `keys()` function.

```
print(peirce.keys())
# displays: ['person_id','name','details']
```

5.4.2. Iterating over a list

When the data includes a `list`, it may be necessary to *iterate* over the list to access individual values. This is done using a `for in` loop.

```
# A list of pragmatic philosophers
pragmatists = ["Charles Sanders Peirce","William James","John Dewey"]

for philosopher in pragmatists:
    print("Pragmatist:",philosopher)

# displays:
# Pragmatist: Charles Sanders Peirce
# Pragmatist: William James
# Pragmatist: John Dewey
```

When iterating over a loop, it is also possible to get the index of the item in the list. This is done by using the `enumerate function`.

```
# A list of pragmatic philosophers
pragmatists = ["Charles Sanders Peirce","William James","John Dewey"]

for idx,philosopher in enumerate(pragmatists):
    print(f"Pragmatist {idx}:",philosopher)

# displays:
# Pragmatist 0: Charles Sanders Peirce
# Pragmatist 1: William James
# Pragmatist 2: John Dewey
```

5.4.3. Accessing values of a list

To access a specific value of a list, the `index` (position) of that value in the list can be used. Note that indexes in Python start with `0`.

```
pragmatists = ["Charles Sanders Peirce","William James","John Dewey"]

# Get james via index
james = pragmatists[1]

print(james) #displays: William James
```

5.4.4. Combining dicts and lists

Often, a combination of techniques will need to be used to access data:

```
#A list of dicts:
pragmatists = [{"name":"Charles"},{"name":"William"},{"name":"John"}]

#Iterate over the list and get value via key
for person in pragmatists:
    person_name = person['name']
    print(person_name)

# Displays:
# Charles
# William
# John
```

```
# A dict with lists
philosophers = {
    "pragmatism":["Peirce","James","Dewey"],
    "idealism":["Plato","Kant","Hegel"]
}

# iterate over idealists
for idealist in philosophers['idealism']:
    print(idealist)

# displays
# Plato
# Kant
# Hegel
```

```
# iterate over the whole structure
for school in philosophers.keys():
    print("Philosophical School:": school)
    names = philosophers[school]
    for name in names:
        print("> Philosopher:",name)

# Displays:
# Philosophical School: pragmatism
# > Philosopher: Peirce
# > Philosopher: James
# > Philosopher: Dewey
# Philosophical School: idealism
# > Philosopher: Plato
# > Philosopher: Kant
# > Philosopher: Hegel
```

5.5. Additional exemplars

5.5.1. Modifying values

Round a float to a given level of precision:

```
pi = 3.1415926
round(pi,2) # displays 3.14
```

5.5.2. Sets

Sets are like lists, but they are unordered and only have 1 of each item.

To add an item:

```
set1 = {1,2}
set1.add(3)
set1 #displays: {1,2,3}
```

To remove an item:

```
set2 = {3,4,5,6}
set2.remove(6)
set2 # displays: {3,4,5}
```

The union of 2 sets:

```
set1.union(set2) #displays {1,2,3,4,5}
```

The intersection of 2 sets:

```
set1.intersection(set2) # displays: {3}
```

The difference between sets:

```
set2 - set1 #displays: {4,5}
```

6 Visualising

6.1. Displaying data within a string

Python allows you to format a string with variables. When a variable is placed inside curly brackets `{}` in a formatted string (`f"???"`), the contents of the variable are rendered as a string. The pattern used for reading and writing files uses formatted strings:

```
file_path = "data/"
file_name = "my_file.txt"
formatted_string = f"{file_path}{file_name}
print(formatted_string)
# displays: 'data/my_file.txt'
```

Numerical values can be modified for display purposes by adding a formatter following a colon directly after the variable name:

Rounding formatter:

```
pi = 3.1415926
f"The value of pi is {pi:.2} (accurate to 2 decimal places)"
# displays: 'The value of pi is 3.14 (accurate to 2 decimal places)'
```

Percent formatter:

```
ratio = 0.3456
f"Ratio as a percent is {ratio:.1%} (1 decimal place)"
# displays: 'Ratio as a percent is 34.6% (1 decimal place)'
```

Comma formatter:

```
population = 2416327
f"Current population: {population:,}"
# displays: 'Current population: 2,416,327
```

6.2. Simple dataframe plots

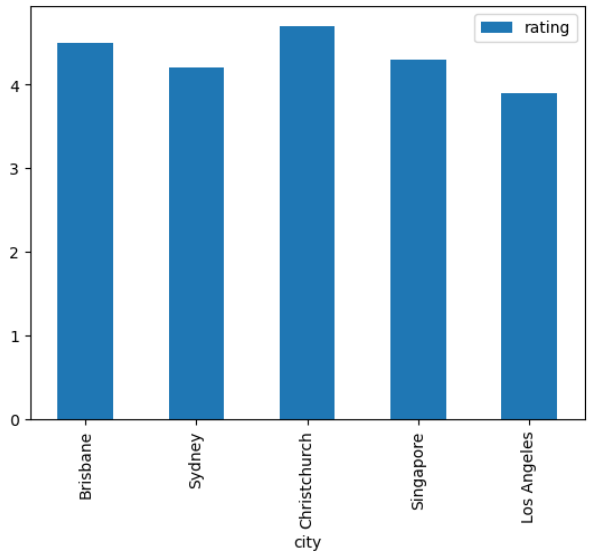
One of the simplest ways of visualising data that is in a pandas dataframe is just to call the `plot()` method of the dataframe.

To see how this works, we first need to create a dataframe.

```
import pandas as pd
data = {"city":['Brisbane','Sydney','Christchurch','Singapore','Los Angeles'],
"rating":[4.5,4.2,4.7,4.3,3.9]}
df = pd.DataFrame(data).set_index('city')
print(df)
```

```
              rating
city
Brisbane         4.5
Sydney           4.2
Christchurch     4.7
Singapore        4.3
Los Angeles      3.9
```

```
chart = df.plot(kind='bar')
```



6.3. Charts with Plotly

Import the relevant plotly library:

```
# import the plotly express library
import plotly.express as px
```

6.3.1. Box plots

Simple box plot:

```
fig = px.box(df,x='col1')
fig.show()
```

Multiple box plots based on categorical data:

```
fig = px.box(df,x='col1',color='categorical_col')
fig.show()
```

6.3.2. Histogram

Simple histogram:

```
fig = px.histogram(df['col'])
fig.show()
```

Specify number of bins, add a boxplot, and label the counts:

```
fig = px.histogram(df['col'],nbins=5,marginal="box",text_auto=True)
fig.show()
```

Show multiple columns as a stacked histogram with color set by categorical data:

```
fig = px.histogram(df[['col1','col2']],nbins=5,marginal="box",text_auto=True,color="categorical_column")
fig.show()
```

7 Sharing

The most common approach to sharing data is to write it to a file. In many instances, writing data to a file is very similar to reading data from a file (see chapter **??**)...

7.1. Sharing structured data

7.1.1. Writing a datafrmae to a CSV file

The `pandas` library provides a function for writing (saving) CSV files directly from a dataframe to a file. Each of the following examples requires the library, so the following `import` needs to be included prior to any of these examples.

```
import pandas as pd
```

The simplest `pandas.to_csv()` function writes a dataframe to a file including the columns and index.

```
# Write a dataframe to a CSV file
file_path = "data/"
file_name = "my_data.csv"
df.to_csv(f"{file_path}{file_name}")
```

To check the file has written corrected, open the file and look at its contents. You can also verify by reading the file into a new dataframe and comparing the results...

```
check_df = pd.read_csv(f"{path}/{file_name}")
```

7.1.2. Writing a dataframe to an Excel file

Pandas can write dataframes to other formats, such as excel `.to_excel()`. Ensure that the file extension is set correctly. Pandas uses the newer `.xlsx` format, not the older `.xls` format.

```
# Make sure the file extension .xlsx matches the save format
# The old excel format of .xls does not work with the current version of pandas!
file_path = "data/"
excel_file = "my_data.xlsx" # <---- note .xlsx
df.to_excel(f"{file_path}/{excel_file}")
```

It is sensible to check the file by opening in Excel to ensure that it has been saved correctly.

Options include specifying the sheet name as well as whether to write the index. Other options can be found in the documentation.

```
file_path = "data/"
excel_file = "my_data.xlsx"
df.to_excel(f"{file_path}/{excel_file}",sheet_name = "my_sheet",index=False)
```

7.2. Sharing semi-structured data

7.2.1. Writing json files

```
# The json library is necessary
import json

# Start with a dict or list of dicts
philosophers = {
"pragmatism":["Peirce","James","Dewey"],
"idealism":["Plato","Kant","Hegel"]
}

# dump the dict to a json string
json_string = json.dumps(philosophers)

# write the string to a file
file_path = "data/"
file_name = "philosophers.json"

with open(f"{file_path}{file_name}",'w') as fp:
fp.write(json_string)
```