Python_for_data

January 6, 2024

1 Notebook for Data Analytics in Python

Python was not written with data analysis in mind. It's a general purpose programing language!

This notebook is similar to Rmarkdown, but less practical:-) Knitting a Jupyter Notebook to a PDF is a multi-step process that typically involves converting the notebook to a different format first (like HTML or LaTeX) and then to a PDF. Here's a general process you can follow:

- `nbconvert` is a tool provided by Jupyter that allows you to convert notebooks to various other formats, including HTML and PDF. First, ensure that `nbconvert` is installed. Run this command in your terminal:`conda install nbconvert`
- To convert a notebook to PDF, you need a TeX distribution installed. nbconvert uses TeX to generate PDFs. For Windows, you can use MikTeX. For macOS, MacTeX is a common choice.
- `jupyter nbconvert --to pdf YourNotebook.ipynb`
- `jupyter nbconvert --to html YourNotebook.ipynb`

2 Introduction

The following command instructs Python to join together the numbers 3, 4, and 5, and to save them as a list named x. When we type x, it gives us back the list.b

[1]:
```python
x = [3, 4, 5]
x
```

[1]:
```
[3, 4, 5]
```

Note that we used the brackets [] to construct this list. We will often want to add two sets of numbers together. It is reasonable to try the following code, though it will not produce the desired results.

[2]:
```python
y = [4, 9, 7]
x + y
```

[2]:
```
[3, 4, 5, 4, 9, 7]
```

The result may appear slightly counterintuitive: why did Python not add the entries of the lists element-by-element? In Python, lists hold arbitrary objects, and are added using concatenation.

In fact, concatenation is the behavior that we saw earlier when we entered “hello” + ” ” + “world”. This example reflects the fact that Python is a general-purpose programming language. Much of Python’s data-specific functionality comes from other packages, notably `numpy` and `pandas`. We’ll see a lot but here is the starter:

[1]:
```
import numpy as np
x = np.array([3, 4, 5])
y = np.array([4, 9, 7])
x + y
```

[1]:
```
array([ 7, 13, 12])
```

In `numpy`, matrices are typically represented as two-dimensional arrays, and vectors as one-dimensional arrays. We’ll see `numpy` later in Cahpter 9 more detail. Let’s start with simple examples without calling a library

[ ]:
```
Create two variables named `a` and `b` so that they contain the following
↪strings respectively: `23 to 0` and `C'est la piquette, Jack!`.
```

[1]:
```
a = "23 to 0"
b = "C'est la piquette, Jack!"
```

[2]:
```
print(a)
print(b)
```

```
23 to 0
C'est la piquette, Jack!
```

Display the number of characters from a, then b.

[25]:
```
print(len(a))
print(len(b))
```

```
7
24
```

Concatenate a and b in a single string, adding a comma as a separating character.

[29]:
```
print(a + ", "+ b)
```

```
23 to 0, C'est la piquette, Jack!
```

Same question by choosing a separation that allows a line break between the two sentences.

[32]:
```
print(a + "\n" + b)
```

```
23 to 0
C'est la piquette, Jack!
```

Using the appropriate method, capitalize a and b.

[42]:
```python
a = a.upper()
b = b.upper()
print(a)
print(b)
```

```
23 TO 0
C'EST LA PIQUETTE, JACK!
```

Using the appropriate method, lowercase a and b.

[44]:
```python
a = a.lower()
b = b.lower()
print(a)
print(b)
```

```
23 to 0
c'est la piquette, jack!
```

Extract the word la and Jack from the string b, using indexes.

[58]:
```python
#la
ind = b.find("la")
l = len("la")
print(b[ind:ind + l])
#Jack
ind = b.find("jack")
l = len("jack")
print(b[ind:ind + l])
```

```
la
jack
```

Look for the sub-chain piqu in b, then do the same with the sub-chain mauvais.

3 Types of Data

`str()`, `int()`, `float()`, `date()`

[63]:
```python
x = 1.5
print(type(x))
print(int(x))
print(x == 1)
```

```
<class 'float'>
1
False
```

4 Structures

4.1 Lists

[64]:
```
x = ["Pascaline", "Gauthier", "Xuan", "Jimmy"]
print(x)
```

```
['Pascaline', 'Gauthier', 'Xuan', 'Jimmy']
```

[65]:
```
z = ["Piketty", "Thomas", 1971]
print(z)
```

```
['Piketty', 'Thomas', 1971]
```

[67]:
```
print(x[0]) # The first element of x
print(x[-1]) # The last element of x
```

```
Pascaline
Jimmy
```

[68]:
```
print(x[1:2]) # The first and second elements of x
print(x[2:]) # From the second element (not included) to the end of x
print(x[:-2]) # From the first to the penultimate (not included)
```

```
['Gauthier']
['Xuan', 'Jimmy']
['Pascaline', 'Gauthier']
```

[77]:
```
tweets = ["aaa", "bbb"]
followers = ["Anne", "Bob", "Irma", "John"]
conuts = [tweets, followers]
res = conuts[1][3] # The 4th item of the 2nd item on the list counts
print(res)
```

```
John
```

[80]:
```
print(len(conuts))
print(len(conuts[1]))
```

```
2
4
```

4.1.1 Modifications

[81]:
```
x = [1, 3, 5, 6, 9]
x[3] = 7 # Replacing the 4th element
print(x)
```

```
[1, 3, 5, 7, 9]
```

```
[83]: x.append(11) # Add value 11 at the end of the list
print(x)
y = [13, 15]
x.extend(y)
print(x)
```

```
[1, 3, 5, 7, 9, 11, 11]
[1, 3, 5, 7, 9, 11, 11, 13, 15]
```

```
[84]: x.remove(3) # Remove the fourth element
print(x)
x = [1, 3, 5, 6, 9]
del x[3] # Remove the fourth element
print(x)
```

```
[1, 5, 7, 9, 11, 11, 13, 15]
```

```
[85]: x = [1, 3, 5, 6, 10]
x[3:5] = [7, 9] # Replaces 4th and 5th values
print(x)
```

```
[1, 3, 5, 7, 9]
```

```
[86]: x = [1, 2, 3, 4, 5]
x[2:3] = ['a', 'b', 'c', 'd'] # Replaces the 3rd value
print(x)
```

```
[1, 2, 'a', 'b', 'c', 'd', 4, 5]
```

Verifying if a Value is Present

```
[87]: x = [1, 2, 3, 4, 5]
print(1 in x)
```

```
True
```

Be careful, copying a list is not trivial in Python. Let's take an example.

```
[88]: x = [1, 2, 3]
y = x
y[0] = 0
print(y)
print(x)
```

```
[0, 2, 3]
[0, 2, 3]
```

To copy a list, there are several ways to do so. Among them, the use of the list() function:

[89]:
```
x = [1, 2, 3]
y = list(x)
y[0] = 0
print("x : ", x)
print("y : ", y)
```

```
x :  [1, 2, 3]
y :  [0, 2, 3]
```

It can be noted that when a splitting is done, a new object is created, not a reference:

[90]:
```
x = [1, 2, 3, 4]
y = x[:2]
y[0] = 0
print("x : ", x)
print("y : ", y)
```

```
x :  [1, 2, 3, 4]
y :  [0, 2]
```

4.1.2 Sorting

[1]:
```
x = [2, 1, 4, 3]
x.sort()
print(x)
```

```
[1, 2, 3, 4]
```

4.2 Tuples

The tuples are sequences of Python objects. To create a tuple, one lists the values, separated by commas Unlike lists, tuplets are inalterable (i.e. cannot be modified after they have been created)

[92]:
```
x = 1, 4, 9, 16, 25
print(x)
x[0] = 1
```

```
(1, 4, 9, 16, 25)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[92], line 3
      1 x = 1, 4, 9, 16, 25
      2 print(x)
----> 3 x[0] = 1

TypeError: 'tuple' object does not support item assignment
```

4.3 Sets

Sets are unordered collections of unique elements. The sets are unalterable, not indexed. To create a set, Python provides the set() function. One or more elements constituting the set are provided, separated by commas and surrounded by braces. During the creation of a set, if there are duplicates in the values provided, these will be deleted to keep only one value

[93]:
```
new_set = set({"Marseille", "Aix-en-Provence", "Nice", "Rennes"})
print(new_set)
```

```
{'Nice', 'Marseille', 'Aix-en-Provence', 'Rennes'}
```

Equivalently, rather than using the set() function, the set can only be defined using the brackets:

[ ]:
```
new_set = {"Marseille", "Aix-en-Provence", "Nice", "Rennes"}
print(new_set)
```

[94]:
```
new_set = set({"Marseille", "Aix-en-Provence", "Nice", "Marseille", "Rennes"})
print(new_set)
```

```
{'Nice', 'Marseille', 'Rennes', 'Aix-en-Provence'}
```

If the element is already present, it will not be added:

[95]:
```
new_set.add("Rennes")
print(new_set)
```

```
{'Nice', 'Marseille', 'Rennes', 'Aix-en-Provence'}
```

[96]:
```
new_set.add("Toulon")
print(new_set)
```

```
{'Nice', 'Toulon', 'Aix-en-Provence', 'Marseille', 'Rennes'}
```

[97]:
```
new_set.remove("Toulon")
print(new_set)
```

```
{'Nice', 'Aix-en-Provence', 'Marseille', 'Rennes'}
```

[98]:
```
print("Marseille" in new_set)
```

```
True
```

[99]:
```
new_set = set({"Marseille", "Aix-en-Provence", "Nice"})
y = new_set.copy()
y.add("Toulon")
print("y : ", y)
```

```
y :  {'Nice', 'Toulon', 'Marseille', 'Aix-en-Provence'}
```

Conversion to a List One of the interests of sets is that they contain unique elements. Also, when you want to obtain the distinct elements of a list, it is possible to convert it into a set (with the set() function), then to convert the set into a list (with the list() function):

[102]:
```
my_list = ["Marseille", "Aix-en-Provence", "Marseille", "Marseille"]
print(my_list)
```

```
['Marseille', 'Aix-en-Provence', 'Marseille', 'Marseille']
```

[103]:
```
my_set = set(my_list)
print(my_set)
```

```
{'Marseille', 'Aix-en-Provence'}
```

[104]:
```
my_new_list = list(my_set)
print(my_new_list)
```

```
['Marseille', 'Aix-en-Provence']
```

4.4 Dictionaries

Python dictionaries are an implementation of key-value objects, the keys being indexed. Keys are often text, values can be of different types and structures. To create a dictionary, you can proceed by using braces ({}).

[106]:
```
my_dict = { "nom": "Kyrie",
"prenom": "John",
"naissance": 1992,
"equipes": ["Cleveland", "Boston"]}
print(my_dict)
print(my_dict["prenom"])
print("age" in my_dict)
```

```
{'nom': 'Kyrie', 'prenom': 'John', 'naissance': 1992, 'equipes': ['Cleveland',
'Boston']}
John
False
```

[108]:
```
the_keys = my_dict.keys()
print(the_keys)
the_keys_list = list(the_keys)
print(the_keys_list)
```

```
dict_keys(['nom', 'prenom', 'naissance', 'equipes'])
['nom', 'prenom', 'naissance', 'equipes']
```

[109]:
```
the_values = my_dict.values()
print(the_values)
```

```
dict_values(['Kyrie', 'John', 1992, ['Cleveland', 'Boston']])
```

5 Operators

To raise a number to a given power, we use two stars (**) the inclusion tests are performed using the operator in.

[110]:
```
print(3 in [1,2, 3])
print(4 not in [1,2, 3])
dictionnaire = {"nom": "Rockwell", "prenom": "Criquette"}
"age" not in dictionnaire.keys()
```

```
True
True
```

[110]:
```
True
```

And, Or

[111]:
```
x = True
y = False
print(x and y)

x = True
y = True
print(x and y)
```

```
False
True
```

Look lsit of math and stat functions

6 Loading and Saving Data

When we launch Jupyter Notebook, a tree structure is displayed, and we navigate inside it to create or open a notebook. The directory containing the notebook is the current directory. When Python is told to import data (or export objects), the origin (or destination) will be indicated relatively in the current directory, unless absolute paths (i.e., a path from the root /) are used.

If a Python program is started from a terminal, the current directory is the directory in which the terminal is located at the time the program is started.

[114]:
```
import os
cwd = os.getcwd()
print(cwd)
os.listdir()
```

```
/Users/yigitaydede/Library/CloudStorage/Dropbox/Python
```

[114]:
```
['PythR.Rmd',
 'try.py',
 '.Rhistory',
 'Untitled.ipynb',
 'PythR_v2.pdf',
 'PythR_v2.Rmd',
 'PythR.html',
 'StatisticsMachineLearningPython.pdf',
 '.ipynb_checkpoints',
 'flights.csv']
```

[118]:
```python
path = "flights.csv"
my_file = open(path, mode = "r")
print(my_file.read())
```

```
IOPub data rate exceeded.
The notebook server will temporarily stop sending output
to the client in order to avoid crashing it.
To change this limit, set the config variable
`--NotebookApp.iopub_data_rate_limit`.

Current values:
NotebookApp.iopub_data_rate_limit=1000000.0 (bytes/sec)
NotebookApp.rate_limit_window=3.0 (secs)
```

It is important to remember to close the file once we have finished using it. To do this, we use the close() method.

[119]:
```python
my_file.close()
```

A common practice in Python is to open a file in a with block. The reason for this choice is that a file opened in such a block is automatically closed at the end of the block.

[120]:
```python
path = "flights.csv"
num_lines = 5  # Number of lines you want to print (similar to head in R)

with open(path, mode="r") as my_file:
    for i in range(num_lines):
        line = my_file.readline()
        print(line.strip())  # strip() removes trailing newlines or spaces
```

```
YEAR,MONTH,DAY,DAY_OF_WEEK,AIRLINE,FLIGHT_NUMBER,TAIL_NUMBER,ORIGIN_AIRPORT,DEST
INATION_AIRPORT,SCHEDULED_DEPARTURE,DEPARTURE_TIME,DEPARTURE_DELAY,TAXI_OUT,WHEE
LS_OFF,SCHEDULED_TIME,ELAPSED_TIME,AIR_TIME,DISTANCE,WHEELS_ON,TAXI_IN,SCHEDULED
_ARRIVAL,ARRIVAL_TIME,ARRIVAL_DELAY,DIVERTED,CANCELLED,CANCELLATION_REASON,AIR_S
YSTEM_DELAY,SECURITY_DELAY,AIRLINE_DELAY,LATE_AIRCRAFT_DELAY,WEATHER_DELAY
2015,1,1,4,AS,98,N407AS,ANC,SEA,0005,2354,-
11,21,0015,205,194,169,1448,0404,4,0430,0408,-22,0,0,,,,,,
```

```
2015,1,1,4,AA,2336,N3KUAA,LAX,PBI,0010,0002,-
8,12,0014,280,279,263,2330,0737,4,0750,0741,-9,0,0,,,,,,
2015,1,1,4,US,840,N171US,SFO,CLT,0020,0018,-
2,16,0034,286,293,266,2296,0800,11,0806,0811,5,0,0,,,,,,
2015,1,1,4,AA,258,N3HYAA,LAX,MIA,0020,0015,-
5,15,0030,285,281,258,2342,0748,8,0805,0756,-9,0,0,,,,,,
```

In this code:

- `open(path, mode="r")` opens the file in read mode.
- `with` is used for context management. It handles opening and closing the file properly, even if an error occurs.
- `readline()` reads one line from the file each time it is called.
- The `for` loop runs `num_lines` times (5 in this case), each time printing a line from the file.
- `line.strip()` is used to remove any extra newline characters or spaces at the end of each line for cleaner output.

This code will print the first 5 lines of the file "flights.csv". You can adjust num_lines to print as many lines as you need.

In Python, the most common way to import a CSV file as a DataFrame (which is similar to a data.frame in R) is to use the `pandas` library. Pandas is a powerful data manipulation and analysis tool, and it provides a DataFrame object that is quite similar to R's data.frame.

Main Values for How to Open Files. r: Opening to read (default), w:Opening to write, x:Opening to create a document, fails if the file already exists, a: Opening to write, adding at the end of the file if it already exists, +: Opening for update (read and write) To be added to an opening mode for binary files (rb or wb), t:Text mode (automatic decoding of bytes in Unicode). Default if not specified (adds to the mode, like b)

[121]:
```python
import pandas as pd

path = "flights.csv"
df = pd.read_csv(path)

# Now you can work with the DataFrame 'df' just like a data.frame in R
```

```
/var/folders/wt/4xtk6v051vd349k3wktfld480000gn/T/ipykernel_14181/1726951512.py:4
: DtypeWarning: Columns (7,8) have mixed types. Specify dtype option on import
or set low_memory=False.
  df = pd.read_csv(path)
```

The warning you're encountering, `DtypeWarning: Columns (7,8) have mixed types. Specify dtype option on import or set low_memory=False`, is raised by pandas when it detects columns in your CSV file that have mixed data types. This can happen if, for example, a column contains both numbers and strings.

If you're okay with pandas inferring the data types, and the mixed types don't pose a problem for your analysis, you can choose to ignore this warning. After loading the data, if you find that certain columns need to be a different data type, you can convert them using the `astype()` method.

[122]:
```python
print(df.head())
```

```
   YEAR  MONTH  DAY  DAY_OF_WEEK AIRLINE  FLIGHT_NUMBER TAIL_NUMBER  \
0  2015      1    1            4      AS             98      N407AS
1  2015      1    1            4      AA           2336      N3KUAA
2  2015      1    1            4      US            840      N171US
3  2015      1    1            4      AA            258      N3HYAA
4  2015      1    1            4      AS            135      N527AS

  ORIGIN_AIRPORT DESTINATION_AIRPORT  SCHEDULED_DEPARTURE  …  ARRIVAL_TIME  \
0            ANC                 SEA                    5  …         408.0
1            LAX                 PBI                   10  …         741.0
2            SFO                 CLT                   20  …         811.0
3            LAX                 MIA                   20  …         756.0
4            SEA                 ANC                   25  …         259.0

   ARRIVAL_DELAY  DIVERTED  CANCELLED  CANCELLATION_REASON  AIR_SYSTEM_DELAY  \
0          -22.0         0          0                  NaN               NaN
1           -9.0         0          0                  NaN               NaN
2            5.0         0          0                  NaN               NaN
3           -9.0         0          0                  NaN               NaN
4          -21.0         0          0                  NaN               NaN

   SECURITY_DELAY  AIRLINE_DELAY  LATE_AIRCRAFT_DELAY  WEATHER_DELAY
0             NaN            NaN                  NaN            NaN
1             NaN            NaN                  NaN            NaN
2             NaN            NaN                  NaN            NaN
3             NaN            NaN                  NaN            NaN
4             NaN            NaN                  NaN            NaN

[5 rows x 31 columns]
```

When using `pandas.read_csv()` to read a CSV file in Python, you don't need to specify a file mode like 'r' for read or 'w' for write, as you would when using Python's built-in `open()` function. The `read_csv()` function from pandas is specifically designed for reading CSV files, and it handles the file opening and reading internally. It always opens the file in the appropriate mode for reading.

Import from the Internet To import a text file from the Internet, methods from the urllib library can be used:

[125]:
```python
import urllib
from urllib.request import urlopen
url = "http://egallic.fr/Enseignement/Python/fichiers_exemples/fichier_texte.
 ↪txt"
with urllib.request.urlopen(url) as my_file:
    data = my_file.read()
print(data)
```

```
b"Bonjour, je suis un fichier au format txt.\nJe contiens plusieurs lignes,
l'id\xc3\xa9e \xc3\xa9tant de montrer comment fonctionne l'importation d'un tel
fichier dans Python.\nTrois lignes devraient suffir."
```

[126]:
```
print(data.decode())
```

```
Bonjour, je suis un fichier au format txt.
Je contiens plusieurs lignes, l'idée étant de montrer comment fonctionne
l'importation d'un tel fichier dans Python.
Trois lignes devraient suffir.
```

CSV Many databases export their data to CSV (e.g., World Bank, FAO, Eurostat, etc.). To import them into Python, you can use the csv module. Again, we use the `open()` function, with the parameters described earlier

[129]:
```
import csv
path = "flights.csv"
with open(path) as my_file:
    my_file_reader = csv.reader(my_file, delimiter=',', quotechar='"')
    data = [x for x in my_file_reader]

# Print the first five observations
for row in data[:5]:
    print(row)
```

```
['YEAR', 'MONTH', 'DAY', 'DAY_OF_WEEK', 'AIRLINE', 'FLIGHT_NUMBER',
'TAIL_NUMBER', 'ORIGIN_AIRPORT', 'DESTINATION_AIRPORT', 'SCHEDULED_DEPARTURE',
'DEPARTURE_TIME', 'DEPARTURE_DELAY', 'TAXI_OUT', 'WHEELS_OFF', 'SCHEDULED_TIME',
'ELAPSED_TIME', 'AIR_TIME', 'DISTANCE', 'WHEELS_ON', 'TAXI_IN',
'SCHEDULED_ARRIVAL', 'ARRIVAL_TIME', 'ARRIVAL_DELAY', 'DIVERTED', 'CANCELLED',
'CANCELLATION_REASON', 'AIR_SYSTEM_DELAY', 'SECURITY_DELAY', 'AIRLINE_DELAY',
'LATE_AIRCRAFT_DELAY', 'WEATHER_DELAY']
['2015', '1', '1', '4', 'AS', '98', 'N407AS', 'ANC', 'SEA', '0005', '2354',
'-11', '21', '0015', '205', '194', '169', '1448', '0404', '4', '0430', '0408',
'-22', '0', '0', '', '', '', '', '', '']
['2015', '1', '1', '4', 'AA', '2336', 'N3KUAA', 'LAX', 'PBI', '0010', '0002',
'-8', '12', '0014', '280', '279', '263', '2330', '0737', '4', '0750', '0741',
'-9', '0', '0', '', '', '', '', '', '']
['2015', '1', '1', '4', 'US', '840', 'N171US', 'SFO', 'CLT', '0020', '0018',
'-2', '16', '0034', '286', '293', '266', '2296', '0800', '11', '0806', '0811',
'5', '0', '0', '', '', '', '', '', '']
['2015', '1', '1', '4', 'AA', '258', 'N3HYAA', 'LAX', 'MIA', '0020', '0015',
'-5', '15', '0030', '285', '281', '258', '2342', '0748', '8', '0805', '0756',
'-9', '0', '0', '', '', '', '', '', '']
```

The two code snippets illustrate two different methods of reading a CSV file in Python: one using the built-in csv module (just above) and the other using the `pandas` library. Both methods achieve the same basic goal — reading data from a CSV file — but they have some key differences in terms of ease of use, functionality, and the data structure they return. `pandas.read_csv` returns a DataFrame, a powerful data structure that provides a lot of functionalities for data manipulation and analysis.

Excel Files We can use the pandas library to read Excel files in Python. Pandas provides the `read_excel()` function, which is specifically designed for this purpose. This function can handle various Excel formats such as .xlsx, .xlsm, .xlsb, and .xls. However, to read Excel files, we need to have the `openpyxl` (for .xlsx files) or xlrd (for older .xls files) packages installed. You can install these packages via pip:

`pip install openpyxl` and `pip install xlrd`

[136]:
```
df2 = pd.read_excel("660218A.xlsx", header = 0)
print(df2.head())
```

```
                          NAME         ID    50  /100  50.1  /100.1    40  \
0  Alimohammadi Sagvand, Sanaz  A00422960  49.0    98  47.0      94  31.0
1                 Biao, Yanan  A00429012  48.5    97  48.0      96  38.0
2                   Cai, Ying  A00431323  49.5    99  49.5      99  28.5
3                Cao, Hanmeng  A00406435  50.0   100  49.0      98  34.5
4                  Cao, Hanyu  A00398061  46.0    92  47.0      94  39.0

   /100.2    60     /100.3  A25MT30F45  A25F75 Letter Mark
0   77.50  46.0  76.666667      81.750  81.500          A-
1   95.00  57.0  95.000000      95.375  95.375          A+
2   71.25  46.0  76.666667      80.625  82.250          A-
3   86.25  59.5  99.166667      95.250  99.125          A+
4   97.50  58.0  96.666667      96.000  95.750          A+
```

Exporting In Python, especially in data science and analytics, the most common file types for saving work depend on the nature of the work and the requirements for data storage, sharing, and interoperability. Here are some of the most commonly used file types:

1. CSV (Comma-Separated Values): Widely used for storing tabular data. Easy to read and write, and can be opened by most spreadsheet programs like Microsoft Excel, Google Sheets, etc. Ideal for relatively simple datasets without complex structures.
2. Excel Files (.xlsx or .xls): Commonly used when sharing data with non-technical stakeholders or when additional formatting and data organization (like multiple sheets) are required.
3. Pickle Files (.pkl): Python-specific binary format. Used for saving Python objects, including DataFrames, preserving their data types and index structures. Not human-readable and not suitable for sharing data across different programming environments.
4. JSON (JavaScript Object Notation): Useful for storing data in a structured, hierarchical format. Human-readable and commonly used for web data and API interactions. Good for data that fits well into a nested, key-value structure.
5. Parquet and Feather Formats: Efficient, columnar storage formats. Ideal for large datasets, and used in big data processing and analytics. Feather is particularly useful for data interchange between Python and R.
6. HDF5 (Hierarchical Data Format version 5): Suitable for storing large quantities of scientific data. Supports data compression and can handle complex data structures.

7. SQL Databases: For projects involving relational databases, saving data directly to SQL databases (like SQLite, MySQL, PostgreSQL) is common. Useful for structured data that needs to be queried or joined with other data in a database.
8. Python Scripts (.py) and Jupyter Notebooks (.ipynb): For saving code, analysis, and documentation. Jupyter Notebooks are particularly popular for exploratory data analysis, as they allow you to combine code, text, and visualizations.

Saving your DataFrame `data` (or `df2`) as a CSV file using pandas is straightforward. You can use the `to_csv` method provided by pandas. Here's how you can do it with chuncks (when you have large data, like `data`):

[143]:
```
print(len(df))
```

```
5819079
```

[141]:
```
chunk_size = 500000  # Size of each chunk - this is just an example
for i in range(0, len(data), chunk_size):
    df[i:i+chunk_size].to_csv(f'data_{i}.csv', index=False)
```

When working with data that's been split into multiple files or batches due to its size, and you need to perform search operations or apply conditions to select specific observations, you typically need to run the search or filter algorithm on each batch. Here's a general approach to how this can be done:

1. **Iterate Through Each Chunk:** Loop through each file (chunk) and perform the necessary search or filtering operation on each one. This can be done by loading each chunk into a pandas DataFrame, applying your conditions to filter or search the data, and then either processing the data immediately or storing the results for further use.
2. **Store or Aggregate Results:** As you process each chunk, you can either: Store the filtered/selected data from each chunk into a new file or a database for later use. Aggregate or summarize the data in memory, if feasible.
3. **Combine Results if Necessary:** After processing all chunks, you might end up with multiple smaller datasets (each representing the filtered data from one chunk). These can be combined, if necessary and if memory allows, or can be left as separate files for further processing.

[144]:
```
import glob

# Condition/filter function
def filter_data(df):
    return df[df['DESTINATION_AIRPORT'] == "MIA"]

# List to hold the filtered data from each chunk
filtered_data = []

# Iterate through each chunk
for filename in glob.glob('data_*.csv'):
    chunk = pd.read_csv(filename)
    filtered_chunk = filter_data(chunk)
```

```
    filtered_data.append(filtered_chunk)

# Combine all filtered data into one DataFrame
final_data = pd.concat(filtered_data)
```

```
/var/folders/wt/4xtk6v051vd349k3wktfld480000gn/T/ipykernel_14181/51342753.py:12:
DtypeWarning: Columns (7,8) have mixed types. Specify dtype option on import or
set low_memory=False.
  chunk = pd.read_csv(filename)
/var/folders/wt/4xtk6v051vd349k3wktfld480000gn/T/ipykernel_14181/51342753.py:12:
DtypeWarning: Columns (7,8) have mixed types. Specify dtype option on import or
set low_memory=False.
  chunk = pd.read_csv(filename)
```

[145]:
```
print(final_data.head())
```

```
     YEAR  MONTH  DAY  DAY_OF_WEEK AIRLINE  FLIGHT_NUMBER TAIL_NUMBER  \
305  2015      6    7            7      AA           1324      N3DRAA
417  2015      6    7            7      AA            349      N016AA
481  2015      6    7            7      UA           1615      N17229
730  2015      6    7            7      AA           2471      N3GDAA
834  2015      6    7            7      AA            919      N004AA

    ORIGIN_AIRPORT DESTINATION_AIRPORT  SCHEDULED_DEPARTURE  …  \
305            LGA                 MIA                 2100  …
417            ATL                 MIA                 2110  …
481            EWR                 MIA                 2115  …
730            LAX                 MIA                 2145  …
834            TPA                 MIA                 2155  …

     ARRIVAL_TIME  ARRIVAL_DELAY  DIVERTED  CANCELLED  CANCELLATION_REASON  \
305        2357.0          -20.0         0          0                  NaN
417        2348.0           39.0         0          0                  NaN
481          13.0          -17.0         0          0                  NaN
730         604.0           12.0         0          0                  NaN
834        2238.0          -22.0         0          0                  NaN

     AIR_SYSTEM_DELAY  SECURITY_DELAY  AIRLINE_DELAY  LATE_AIRCRAFT_DELAY  \
305               NaN             NaN            NaN                  NaN
417               0.0             0.0           39.0                  0.0
481               NaN             NaN            NaN                  NaN
730               NaN             NaN            NaN                  NaN
834               NaN             NaN            NaN                  NaN

     WEATHER_DELAY
305            NaN
417            0.0
481            NaN
```

```
730             NaN
834             NaN

[5 rows x 31 columns]
```

If you want to add an additional condition to your filtering process, you can modify the filter_data function to include this new condition. In addition to filtering rows where `DESTINATION_AIRPORT` is “MIA”, you also want to filter based on another condition, such as ‘ORIGIN_AIRPORT’ being “LAX”. You can use the logical AND operator (&) to combine these conditions in `pandas`.

[148]:
```python
# If you're confident that pandas' type inference won't adversely affect your
↪analysis,
# you can choose to ignore this warning.

import warnings
warnings.filterwarnings('ignore', category=pd.errors.DtypeWarning)

def filter_data(df):
    condition1 = df['DESTINATION_AIRPORT'] == "MIA"
    condition2 = df['ORIGIN_AIRPORT'] == "LAX"
    return df[condition1 & condition2]

# List to hold the filtered data from each chunk
filtered_data = []

# Iterate through each chunk
for filename in glob.glob('data_*.csv'):
    chunk = pd.read_csv(filename)
    filtered_chunk = filter_data(chunk)
    filtered_data.append(filtered_chunk)

# Combine all filtered data into one DataFrame
final_data = pd.concat(filtered_data)

print(final_data.head())
```

```
      YEAR  MONTH  DAY  DAY_OF_WEEK AIRLINE  FLIGHT_NUMBER TAIL_NUMBER  \
730   2015      6    7            7      AA           2471      N3GDAA
1166  2015      6    7            7      DL           1168       N3766
1498  2015      6    7            7      AA           1538      N3KHAA
1539  2015      6    8            1      AA            260      N3HMAA
4126  2015      6    8            1      AA             68      N5EAAA

     ORIGIN_AIRPORT DESTINATION_AIRPORT  SCHEDULED_DEPARTURE  …  \
730             LAX                 MIA                 2145  …
1166            LAX                 MIA                 2220  …
1498            LAX                 MIA                 2355  …
1539            LAX                 MIA                   10  …
```

```
4126            LAX                 MIA                 735  …

      ARRIVAL_TIME  ARRIVAL_DELAY  DIVERTED  CANCELLED  CANCELLATION_REASON  \
730          604.0           12.0         0          0                  NaN
1166         621.0           -3.0         0          0                  NaN
1498         742.0          -19.0         0          0                  NaN
1539         803.0          -19.0         0          0                  NaN
4126        1519.0          -32.0         0          0                  NaN

      AIR_SYSTEM_DELAY  SECURITY_DELAY  AIRLINE_DELAY  LATE_AIRCRAFT_DELAY  \
730                NaN             NaN            NaN                  NaN
1166               NaN             NaN            NaN                  NaN
1498               NaN             NaN            NaN                  NaN
1539               NaN             NaN            NaN                  NaN
4126               NaN             NaN            NaN                  NaN

      WEATHER_DELAY
730             NaN
1166            NaN
1498            NaN
1539            NaN
4126            NaN

[5 rows x 31 columns]
```

6.1 Use R Magic in Python Notebooks:

If you primarily use Python but need to run some R code, you can use the `%R` magic command in a Python notebook. This requires the `rpy2` package in Python. Install `rpy2` via `pip install rpy2` Then, at the start of your Python notebook, load the R magic:

[1]:
```
%load_ext rpy2.ipython
```

[11]:
```
%%R
library(tidyverse)

filter_data <- function(df) {
  df %>% filter(DESTINATION_AIRPORT == "MIA", ORIGIN_AIRPORT == "LAX")
}

read_csv_quietly <- function(filename) {
  read_csv(filename, show_col_types = FALSE)
}

# Get the list of files
# list.files() function with the apptern argument is to list files in the CD␣
 ↪that
# matches a specific pattern
```

```
files <- list.files(pattern = "data_.*\\.csv")

# Read, filter, and combine data from each file
final_data <- files %>%
  map(read_csv_quietly) %>%
  map(filter_data) %>%
  bind_rows()

# Display the head of the final data frame
head(final_data)
```

```
# A tibble: 6 × 31
   YEAR MONTH   DAY DAY_OF_WEEK AIRLINE FLIGHT_NUMBER TAIL_NUMBER ORIGIN_AIRPORT
  <dbl> <dbl> <dbl>       <dbl> <chr>           <dbl> <chr>       <chr>
1  2015     1     1           4 AA                258 N3HYAA      LAX
2  2015     1     1           4 AA                115 N3CTAA      LAX
3  2015     1     1           4 AA                306 N859AA      LAX
4  2015     1     1           4 AA                208 N355AA      LAX
5  2015     1     1           4 AA                124 N7BEAA      LAX
6  2015     1     1           4 AA                 28 N358AA      LAX
#  23 more variables: DESTINATION_AIRPORT <chr>, SCHEDULED_DEPARTURE <dbl>,
#   DEPARTURE_TIME <dbl>, DEPARTURE_DELAY <dbl>, TAXI_OUT <dbl>,
#   WHEELS_OFF <dbl>, SCHEDULED_TIME <dbl>, ELAPSED_TIME <dbl>, AIR_TIME <dbl>,
#   DISTANCE <dbl>, WHEELS_ON <dbl>, TAXI_IN <dbl>, SCHEDULED_ARRIVAL <dbl>,
#   ARRIVAL_TIME <dbl>, ARRIVAL_DELAY <dbl>, DIVERTED <dbl>, CANCELLED <dbl>,
#   CANCELLATION_REASON <chr>, AIR_SYSTEM_DELAY <dbl>, SECURITY_DELAY <dbl>,
#   AIRLINE_DELAY <dbl>, LATE_AIRCRAFT_DELAY <dbl>, WEATHER_DELAY <dbl>
```

In the code snippet provided above, `map` is a function from the `purrr` package, which is part of the tidyverse suite of packages in R. The map function is used to apply a function to each element of a list or vector and is particularly useful for performing operations on lists in a concise and readable way.

[18]:
```
%%R

# OR

library(tidyverse)

filter_data <- function(dff){
  dff %>% filter(DESTINATION_AIRPORT == "MIA", ORIGIN_AIRPORT == "LAX")
}

# matches a specific pattern
files <- list.files(pattern = "data_.*\\.csv")

filtered_data <- NULL
```

```r
# Iterate through each chunk
for (filename in files) {
  chunk <- read_csv(filename, show_col_types = FALSE)
  filtered_chunk <- filter_data(chunk)
  filtered_data <- rbind(filtered_data, filtered_chunk)
}

head(as.data.frame(filtered_data))
```

```
  YEAR MONTH DAY DAY_OF_WEEK AIRLINE FLIGHT_NUMBER TAIL_NUMBER ORIGIN_AIRPORT
1 2015     1   1           4      AA           258      N3HYAA            LAX
2 2015     1   1           4      AA           115      N3CTAA            LAX
3 2015     1   1           4      AA           306      N859AA            LAX
4 2015     1   1           4      AA           208      N355AA            LAX
5 2015     1   1           4      AA           124      N7BEAA            LAX
6 2015     1   1           4      AA            28      N358AA            LAX
  DESTINATION_AIRPORT SCHEDULED_DEPARTURE DEPARTURE_TIME DEPARTURE_DELAY
1                 MIA                  20             15              -5
2                 MIA                 105            103              -2
3                 MIA                 800            759              -1
4                 MIA                 840            842               2
5                 MIA                1250           1305              15
6                 MIA                1500           1456              -4
  TAXI_OUT WHEELS_OFF SCHEDULED_TIME ELAPSED_TIME AIR_TIME DISTANCE WHEELS_ON
1       15         30            285          281      258     2342       748
2       14        117            286          276      255     2342       832
3       13        812            292          280      257     2342      1529
4       19        901            293          285      264     2342      1625
5       17       1322            285          277      256     2342      2038
6       19       1515            290          305      284     2342      2259
  TAXI_IN SCHEDULED_ARRIVAL ARRIVAL_TIME ARRIVAL_DELAY DIVERTED CANCELLED
1       8               805          756            -9        0         0
2       7               851          839           -12        0         0
3      10              1552         1539           -13        0         0
4       2              1633         1627            -6        0         0
5       4              2035         2042             7        0         0
6       2              2250         2301            11        0         0
  CANCELLATION_REASON AIR_SYSTEM_DELAY SECURITY_DELAY AIRLINE_DELAY
1                <NA>               NA             NA            NA
2                <NA>               NA             NA            NA
3                <NA>               NA             NA            NA
4                <NA>               NA             NA            NA
5                <NA>               NA             NA            NA
6                <NA>               NA             NA            NA
  LATE_AIRCRAFT_DELAY WEATHER_DELAY
1                  NA            NA
2                  NA            NA
```

```
3                   NA             NA
4                   NA             NA
5                   NA             NA
6                   NA             NA
```

7 Conditions

Often, depending on the evaluation of an expression, one wants to perform one operation rather than another.

7.1 if

if expression: instruction

[19]:
```python
x = 2
if x == 2:
  print("Hello")
```

```
Hello
```

[20]:
```python
# Not this
x=3
if x == 2:
  print("Hello")
```

Inside the block, several instructions can be written that will be evaluated if the expression is True:

[21]:
```python
x=2
if x == 2:
    y = "Hello"
    print(y + ", x is : " + str(x))
```

```
Hello, x is : 2
```

7.2 if-else

if expression: instructions else: other_instruction

[22]:
```python
temperature = 26
heat = ""

if temperature > 28:
    heat = "hot"
else:
    heat = "cold"

print("It is " + heat + " out there")
```

```
It is cold out there
```

7.3 if-elif

[23]:
```python
temperature = -4
heat = ""

if temperature > 28:
    heat = "hot"
elif temperature <= 28 and temperature > 15:
    heat = "not too hot"
elif temperature <= 15 and temperature > 0:
    heat = "cold"
else:
    heat = "very cold"

print("It is " + heat + " out there")
```

```
It is very cold out there
```

8 Loops

When the same operation has to be repeated several times, for a given number of times or as long as a condition is verified (or as long as it is not verified), loops can be used, which is much less painful than evaluating by hand or by copying and pasting the same instruction.

We will discuss two types of loops in this chapter:

- those for which we do not know a priori the number of iterations (the number of repetitions) to be performed: `while()` loops
- those for which we know a priori how many iterations are necessary: `for()` loops

[24]:
```python
x = 100
while x/3 > 1:
    print(x/3)
    x = x/3
```

```
33.333333333333336
11.111111111111112
3.703703703703704
1.234567901234568
```

[28]:
```python
# Single loop
total = 0
for value in [3, 2, 19]:
    total += value

print('Total is: {0}'.format(total))
```

```
Total is: 24
```

We also took advantage of the increment notation in Python: the expression `a += b` increment is equivalent to `a = a + b`. R doesn't have an in-built shorthand operator for incrementing a variable. So, the expression `a += b` that you would use in other languages is written as `a = a + b` in R.

[29]:
```python
# See the difference

total = 0
for value in [3, 2, 19]:
    total += value
    print('Total is: {0}'.format(total))
```

```
Total is: 3
Total is: 5
Total is: 24
```

[30]:
```python
## Double loop
total = 0
for value in [2,3,19]:
    for weight in [3, 2, 1]:
        total += value * weight

print('Total is: {0}'.format(total))
```

```
Total is: 144
```

When we know the number of iterations in advance, we can use a `for()`loop. The syntax is as follows:

[25]:
```python
message = "The squared value of {} is {}"
n = 10
for i in range(0, n + 1):
    print(message.format(i, i ** 2))
```

```
The squared value of 0 is 0
The squared value of 1 is 1
The squared value of 2 is 4
The squared value of 3 is 9
The squared value of 4 is 16
The squared value of 5 is 25
The squared value of 6 is 36
The squared value of 7 is 49
The squared value of 8 is 64
The squared value of 9 is 81
The squared value of 10 is 100
```

If we want to store the result in a list:

[26]:
```
n=10
n_squares = []

for i in range(0, n+1):
    n_squares.append(i**2)

print(n_squares)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

[27]:
```
%%R

n <- 10
n_squares <- c()  # Initialize an empty vector

# R starts at 1 not zero:-)
for (i in 0:n) {
  n_squares <- c(n_squares, i^2)  # Append the square of i to the vector
}

print(n_squares)
```

```
 [1]   0   1   4   9  16  25  36  49  64  81 100
```

[30]:
```
message = "There is(are) {} letter(s) in the name: {}"
for first_name in ["Pascaline", "Gauthier", "Xuan", "Jimmy"]:
  print(message.format(len(first_name), first_name))

# Cleaner

message = "There is(are) {} letter(s) in the name: {}"
for i in ["Pascaline", "Gauthier", "Xuan", "Jimmy"]:
    print(message.format(len(i), i))
```

```
There is(are) 9 letter(s) in the name: Pascaline
There is(are) 8 letter(s) in the name: Gauthier
There is(are) 4 letter(s) in the name: Xuan
There is(are) 5 letter(s) in the name: Jimmy
There is(are) 9 letter(s) in the name: Pascaline
There is(are) 8 letter(s) in the name: Gauthier
There is(are) 4 letter(s) in the name: Xuan
There is(are) 5 letter(s) in the name: Jimmy
```

Here is the R code, where `sprintf()` is used to format the message. It replaces `%s` with the number of characters in the name (`nchar(first_name)`) and the name itself (`first_name`).

[32]:
```
%%R
```

```
message <- "There is(are) %s letter(s) in the name: %s"
first_names <- c("Pascaline", "Gauthier", "Xuan", "Jimmy")

for (i in first_names) {
  formatted_message <- sprintf(message, nchar(i), i)
  print(formatted_message)
}

# OR

message <- "There is(are) %s letter(s) in the name: %s"

for (i in c("Pascaline", "Gauthier", "Xuan", "Jimmy")) {
  formatted_message <- sprintf(message, nchar(i), i)
  print(formatted_message)
}
```

```
[1] "There is(are) 9 letter(s) in the name: Pascaline"
[1] "There is(are) 8 letter(s) in the name: Gauthier"
[1] "There is(are) 4 letter(s) in the name: Xuan"
[1] "There is(are) 5 letter(s) in the name: Jimmy"
[1] "There is(are) 9 letter(s) in the name: Pascaline"
[1] "There is(are) 8 letter(s) in the name: Gauthier"
[1] "There is(are) 4 letter(s) in the name: Xuan"
[1] "There is(are) 5 letter(s) in the name: Jimmy"
```

[35]:
```
# This will be intresting for more complex loops
import time  # Importing time for demonstration purposes

for i in range(3):
    for j in range(3):
        print(f"i equals {i} and j equals {j}", end="\r")
        time.sleep(1)  # Adding a delay for demonstration purposes
```

```
i equals 2 and j equals 2
```

[39]:
```
%%R

# Try it in RStudio!  Doesn't work here:-)

library(progress)

# Define the total number of iterations
total <- 3 * 3  # As we have two loops running from 0 to 2

# Create a progress bar
pb <- progress_bar$new(
```

```
  format = "[:bar] :percent :elapsedfull",
  total = total,
  clear = FALSE,
  width = 60
)

# Iterate with a progress bar
for (i in 0:2) {
  for (j in 0:2) {
    # Update the progress bar
    pb$tick()

    # Your loop content goes here
    # ...

    Sys.sleep(0.1)  # Added only to slow down the loop for demonstration
  }
}
```

[40]:
```python
import ipywidgets as widgets
from IPython.display import display
import time

progress = widgets.IntProgress(value=0, min=0, max=9, description='Loading:')
display(progress)

for i in range(9):
    time.sleep(0.5)  # Simulating work
    progress.value += 1
```

```
IntProgress(value=0, description='Loading:', max=9)
```

We don't need to memorize these, but when we need them, we know they exist!

[41]:
```python
message = "New value for j: {}"
j = 10
for i in range(0, 4):
    j += 5
    print(message.format(j))
```

```
New value for j: 15
New value for j: 20
New value for j: 25
New value for j: 30
```

In a loop, if we want to increment a counter, we can use the symbol **+=** rather than writing 'counter = counter + . . . ". But this is not availbale in R

[42]:
```
%%R

j <- 10
for (i in 1:4) {
    j <- j + 5
    cat("New value for j:", j, "\n")
}
```

```
New value for j: 15
New value for j: 20
New value for j: 25
New value for j: 30
```

Here is a good example: Choose a ‘mystery’ number between 1 and 100, and store it in an object called `mystere_number`. Then, create a loop that at each iteration performs a random draw of an integer between 1 and 100. As long as the number drawn is different from the mystery number, the loop must continue. At the output of the loop, a variable called `nb_drawings` will contain the number of prints made to obtain the mystery number.

[64]:
```
%%R

mystere_number <- sample(1:100, 1)
rd <- 0

while (rd != mystere_number) {
    rd <- sample(1:100, 1)
}

c(mystere_number, nb_drawings)
```

```
[1] 39 39
```

[94]:
```
import random
import ipywidgets as widgets
from IPython.display import display
import time

progress = widgets.IntProgress(value=0, min=0, max=3000, description='Loading:')
display(progress)


mystere_number = random.randint(0, 2000)
rd = 0

while rd != mystere_number:
    rd = random.randint(0, 2000)
    progress.value += 1
```

```
print(mystere_number, rd)
```

```
IntProgress(value=0, description='Loading:', max=3000)
```

```
1746 1746
```

Use a loop to scan integers from 1 to 20 using a for loop, displaying in the console at each iteration if the current number is even.

[96]:
```
message1 = "The value {} is even"
message2 = "The value {} is not even"

for i in range(1, 21):
    if i % 2 == 0:
        print(message1.format(i))
    else:
        print(message2.format(i))
```

```
The value 1 is not even
The value 2 is even
The value 3 is not even
The value 4 is even
The value 5 is not even
The value 6 is even
The value 7 is not even
The value 8 is even
The value 9 is not even
The value 10 is even
The value 11 is not even
The value 12 is even
The value 13 is not even
The value 14 is even
The value 15 is not even
The value 16 is even
The value 17 is not even
The value 18 is even
The value 19 is not even
The value 20 is even
```

[97]:
```
%%R

message1 <- "The value %s is even"
message2 <- "The value %s is not even"

for (i in 1:20) {
  if (i %% 2 == 0) {
    cat(sprintf(message1, i), "\n")
  } else {
    cat(sprintf(message2, i), "\n")
```

```
  }
}

# OR

for (i in 1:20) {
  if (i %% 2 == 0) {
    paste("The value ", i, " is even")
  } else {
    paste("The value ", i, " is not even")
  }
}
```

```
The value 1 is not even
The value 2 is even
The value 3 is not even
The value 4 is even
The value 5 is not even
The value 6 is even
The value 7 is not even
The value 8 is even
The value 9 is not even
The value 10 is even
The value 11 is not even
The value 12 is even
The value 13 is not even
The value 14 is even
The value 15 is not even
The value 16 is even
The value 17 is not even
The value 18 is even
The value 19 is not even
The value 20 is even
```

Perhaps a more common task would be to sum over (`value`, `weight`) pairs. For instance, to compute the average value of a random variable that takes on possible values 2, 3 or 19 with probability 0.2, 0.3, 0.5 respectively we would compute the weighted sum. Tasks such as this can often be accomplished using the `zip()` function that loops over a sequence of tuples.

[31]:
```
total = 0
for value, weight in zip([2,3,19],
                         [0.2,0.3,0.5]):
    total += weight * value
print('Weighted average is: {0}'.format(total))
```

```
Weighted average is: 10.8
```

[33]:
```
%%R

values <- c(2, 3, 19)
weights <- c(0.2, 0.3, 0.5)

total <- 0
for (i in 1:length(values)) {
    total <- total + weights[i] * values[i]
}

cat('Weighted average is:', total, '\n')
```

```
Weighted average is: 10.8
```

Wow, how stu…d it's! A Simple version would be you'd perform a vectorized operation in NumPy

[32]:
```
import numpy as np

values = np.array([2, 3, 19])
weights = np.array([0.2, 0.3, 0.5])

total = np.sum(weights * values)
print('Weighted average is:', total)
```

```
Weighted average is: 10.8
```

[34]:
```
%%R

values <- c(2, 3, 19)
weights <- c(0.2, 0.3, 0.5)

total <- sum(weights * values)

cat('Weighted average is:', total, '\n')
```

```
Weighted average is: 10.8
```

9 FUNCTIONS

Most of the time, we use the basic functions or those contained in modules. However, when retrieving data online or formatting data imported from various sources, it may be necessary to create our own functions.

`def name_function(arguments):      body of the function4`

Once the function is defined, it is called by referring to its name: `name_function()`

[98]:
```
def square(x):
    return x**2
```

```
square(2)
```

[98]: 4

[99]:
```
%%R

squarer <- function(x){
    return(x^2)
}
squarer(2)
```

```
[1] 4
```

In both Python and R, functions can have positional arguments and keyword (named) arguments. However, there are some differences in how these arguments are handled in each language.

Python: **Positional Arguments:** These are arguments that can be called by their position in the function definition. **Keyword Arguments:** Also known as named arguments, these are specified by name. One key feature in Python is that you can have default values for keyword arguments. Example in Python:

[100]:
```python
def greet(name, msg="Hello"):
    print(msg, name)

greet("Alice")              # Uses default value for msg
greet("Bob", "Goodbye")     # Overrides default value for msg
greet(msg="Hi", name="Eve")  # Using named arguments
```

```
Hello Alice
Goodbye Bob
Hi Eve
```

[101]:
```
%%R

greet <- function(name, msg="Hello") {
  cat(msg, name, "\n")
}

greet("Alice")                 # Uses default msg
greet("Bob", "Goodbye")        # Overrides default msg
greet(name="Eve", msg="Hi")    # Named arguments, order can be changed
greet("Charlie", msg="Hey")    # Mix of positional and named arguments
```

```
Hello Alice
Goodbye Bob
Hi Eve
Hey Charlie
```

A function can be provided as an argument to another function.

[103]:
```python
def square(x):
    """Returns the squared value of x"""
    return x**2

def apply_fun_to_4(fun):
    """Applies the function `fun` to 4"""
    return fun(4)

print(apply_fun_to_4(square))
```

```
16
```

[110]:
```r
%%R

# Define the square function
square <- function(x) {
  return(x^2)
}

# Define a function that applies another function to 4
apply_fun_to_4 <- function(fun) {
  return(fun(4))
}

# Apply the square function to 4 and print the result
result <- apply_fun_to_4(square)
cat("Result:", result, "\n")
```

```
Result: 16
```

[ ]:
```
When a function is called, the body of that function is interpreted. Variables
 ↪that have been defined in the body of the function are assigned to a local
 ↪namespace.
In other words, they live only within this local space, which is created at the
 ↪moment of the call of the function and destroyed at the end of it.
```

[106]:
```python
value = 10

def f(x):
    value = 2
    return x + value

f(5)
```

[106]: 7

If a variable is not defined in the body of a function, Python will search in a parent environment.

[108]:
```python
def f(y):
    return y + value

f(5)
```

[108]:
```
15
```

[116]:
```
%%R

# Same for R

value = 10

f <- function(x) {
    value = 2
    return(x + value)
}


# But

k <- function(y) {
    return(y + value)
}

c(f(5), value, k(5))
```

```
[1]  7 10 15
```

Python offers what are called lambdas functions, or anonymous functions. A lambda function has only one instruction whose result is that of the function.

[117]:
```python
def square(x):
    return x**2

## The same function with lamda
square_2 = lambda x: x**2
print(square_2(4))
```

```
16
```

It can sometimes be convenient to return several elements in return for a function. Although the list is a candidate for this feature, it may be better to use a dictionary, to be able to access the values with their key!

[122]:
```python
import statistics
```

```python
def stat_des(x):
    """Returns the mean and standard deviation of `x`"""
    return {"mean": statistics.mean(x), "std_dev": statistics.stdev(x)}

x = [1,3,2,6,4,1,8,9,3,2]
res = stat_des(x)
print(res)

print(res["mean"])
```

```
{'mean': 3.9, 'std_dev': 2.8460498941515415}
3.9
```

[123]:
```
%%R

# Define the function
stat_des <- function(x) {
  return(list(mean = mean(x), std_dev = sd(x)))
}

# Example data
x <- c(1, 3, 2, 6, 4, 1, 8, 9, 3, 2)

# Get the result as a named list
res <- stat_des(x)
print(res)

# Access elements of the list
print(res$mean)  # Access the mean
print(res$std_dev)  # Access the standard deviation
```

```
$mean
[1] 3.9

$std_dev
[1] 2.84605

[1] 3.9
[1] 2.84605
```

10 Introduction to Numpy

An important library for numerical calculations: NumPy (abbreviation of Numerical Python). It is common practice to import NumPy by assigning it the alias np: `import numpy as np`

10.1 Arrays

NumPy offers a popular data structure, arrays, on which calculations can be performed efficiently. In `numpy`,an array is a generic term for a multidimensional set of numbers. We use the `np.array()`function to define x and y, which are one-dimensional arrays, i.e. vectors.

[2]:
```
import numpy as np

x = np.array([3, 4, 5])
y = np.array([4, 9, 7])

x + y
```

[2]:
```
array([ 7, 13, 12])
```

In numpy, matrices are typically represented as two-dimensional arrays, and vectors as one-dimensional arrays. We can create a two-dimensional array as follows.

[3]:
```
x = np.array([[1, 2], [3, 4]])
x
```

[3]:
```
array([[1, 2],
       [3, 4]])
```

The object `x` has several attributes, or associated objects. To access an attribute of `x`, we type `x.attribute`, where we replace attribute with the name of the attribute. For instance, we can access the `ndim` attribute of x as follows.

[6]:
```
print(x.ndim)
print(x.dtype)
```

```
2
int64
```

Why is `x` comprised of integers? This is because we created `x` by passing in exclusively integers to the `np.array()` function. If we had passed in any decimals, then we would have obtained an array of floating point numbers (i.e. real-valued numbers).

[7]:
```
np.array([[1, 2], [3.0, 4]]).dtype
```

[7]:
```
dtype('float64')
```

The array `x` is two-dimensional. We can find out the number of rows and columns by looking at its shape attribute

[8]:
```
x.shape
```

[8]:
```
(2, 2)
```

A method is a function that is associated with an object. For instance, given an array `x`, the expression `x.sum()` sums all of its elements, using the `sum()` method for arrays. The call `x.sum()` automatically provides `x` as the first argument to its `sum()` method.

[9]:
```
x = np.array([1, 2, 3, 4])
x.sum()
```

[9]:
```
10
```

We could also sum the elements of `x` by passing in `x` as an argument to the `np.sum()` function.

[10]:
```
x = np.array([1, 2, 3, 4])
np.sum(x)
```

[10]:
```
10
```

Pay attention here:

[13]:
```
x = np.array([1, 2, 3, 4, 5, 6])
print(x)

x_reshaped = x.reshape((2, 3))
print(x_reshaped)

#Change the first element to 5
x_reshaped[0, 0] = 5

print(x_reshaped)
print(x)
```

```
[1 2 3 4 5 6]
[[1 2 3]
 [4 5 6]]
[5 2 3 4 5 6]
[[5 2 3]
 [4 5 6]]
```

To our surprise, we discover that the first element of x has been modified as well! Modifying `x_reshaped` also modified `x` because the two objects occupy the same space in memory.

10.2 Random data

In data analytics, we often want to generate random data. the `np.random.normal()` function generates a vector of random normal variables. We can learn more about this function by looking at the help page, via a call to `np.random.normal?`. The first line of the help page reads `normal(loc=0.0, scale=1.0, size=None)`. This signature line tells us that the function's arguments are `loc`, `scale`, and `size`. These are keyword arguments, which means that when they are passed into the function, they can be referred to by name (in any order). By default, this function will generate random normal variable(s) with mean (`loc`) 0 and standard deviation (`scale`) 1; further more, a single random variable will be generated unless the argument to size is changed.

```
[14]: x = np.random.normal(size=50)
      x
```

```
[14]: array([-3.98758621e-01, -1.32585520e+00,  2.99086619e-01, -9.00467046e-01,
             -6.24377293e-01,  1.38433653e+00, -1.98727515e+00, -1.33082457e-01,
             -9.71992791e-01,  6.63398551e-01,  3.46517153e-01, -1.63397597e+00,
              3.33672263e-01, -9.34686339e-01, -1.41242974e+00, -1.34432196e-01,
             -1.72489169e+00,  1.60784233e-01, -1.05829487e+00, -2.18321556e-02,
              1.89588479e-01, -1.39836483e-01, -8.93383689e-01, -7.83250154e-02,
              1.17285345e+00, -3.39760151e-01,  7.51434322e-02,  1.19200948e+00,
             -5.65394550e-01, -7.84100601e-01, -9.63238726e-02,  1.05682617e+00,
             -2.19975770e-03, -3.12787509e-01, -4.09556939e-01, -4.53440641e-03,
             -5.26438184e-01, -3.48628682e+00,  5.96224387e-01,  1.29367128e+00,
             -3.66914424e-01,  1.05576017e+00,  1.34325270e+00,  1.01442394e+00,
              1.16487534e+00, -2.99751323e-01,  1.61316775e+00,  3.66836276e-01,
             -2.25382573e-01,  1.15627344e-02])
```

We create an array `y` by adding an independent `N(50,1)` random variable to each element of `x`.

```
[17]: y = x + np.random.normal(loc=50, scale=1, size=50)
      y
```

```
[17]: array([50.91313043, 48.35385782, 50.80903044, 50.54994561, 48.69165226,
             50.60932749, 48.78877271, 50.15383787, 48.42220353, 49.80142401,
             51.68320569, 47.47239158, 50.91925555, 49.34482161, 49.16704937,
             49.48369408, 47.72670698, 50.40858947, 49.02627041, 51.47861593,
             48.97465266, 49.92856398, 50.42015717, 50.57461143, 52.15778099,
             47.33148267, 49.79171346, 51.46796316, 50.55562685, 47.51896059,
             50.43897447, 51.001371  , 51.77533076, 49.44005047, 50.68029124,
             51.48579121, 50.05814013, 47.42541796, 50.95767462, 51.35853017,
             48.78070351, 49.77239821, 50.46416671, 50.91403132, 53.04549545,
             50.24357539, 51.88093265, 50.81059912, 49.5126853 , 51.05077079])
```

In order to ensure that our code provides exactly the same results each time it is run, we can set a random seed using the `np.random.default_rng()`

```
[24]: rng = np.random.default_rng(1303)
      print(rng.normal(scale=5, size=2))

      rng2 = np.random.default_rng(1303)
      print(rng2.normal(scale=5, size=2))
```

```
[ 4.09482632 -1.07485605]
[ 4.09482632 -1.07485605]
```

```
[27]: # Some statistical operations
rng = np.random.default_rng(3)
y = rng.standard_normal(10)
np.mean(y), y.mean()
```

```
[27]: (-0.1126795190952861, -0.1126795190952861)
```

```
[28]: #And
np.var(y), y.var(), np.mean((y - y.mean())**2)
```

```
[28]: (2.7243406406465125, 2.7243406406465125, 2.7243406406465125)
```

10.3 Column/Row operations

The `np.mean()`, `np.var()`, and `np.std()` functions can also be applied to the rows and columns of a matrix. To see this, we construct a 10×3 matrix of N(0,1) random variables, and consider computing its row sums.

```
[29]: X = rng.standard_normal((10, 3))
X
```

```
[29]: array([[ 0.22578661, -0.35263079, -0.28128742],
       [-0.66804635, -1.05515055, -0.39080098],
       [ 0.48194539, -0.23855361,  0.9577587 ],
       [-0.19980213,  0.02425957,  1.54582085],
       [ 0.54510552, -0.50522874, -0.18283897],
       [ 0.54052513,  1.93508803, -0.26962033],
       [-0.24355868,  1.0023136 , -0.88645994],
       [-0.29172023,  0.88253897,  0.58035002],
       [ 0.0915167 ,  0.67010435, -2.82816231],
       [ 1.02130682, -0.95964476, -1.66861984]])
```

```
[30]: X.mean(axis=0)
# Sum of its rows by columns
```

```
[30]: array([ 0.15030588,  0.14030961, -0.34238602])
```

```
[31]: X.mean(axis=1)
# Sum of its columns by rows
```

```
[31]: array([-0.13604387, -0.70466596,  0.40038349,  0.45675943, -0.04765406,
        0.73533095, -0.04256834,  0.39038958, -0.68884708, -0.53565259])
```

10.4 Speed

Arrays are a useful structure for performing basic statistical operations as well as pseudo-random number generation. The structure of the tables is similar to that of the lists, but the latter are slower to process and use more memory. To be convinced:

[19]:
```python
# first lists

from random import random
from operator import truediv
l1 = [random() for i in range(1000)]
l2 = [random() for i in range(1000)]
%timeit s = sum(map(truediv,l1,l2))
```

```
24.6 µs ± 328 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

To write the Python expression `l1 = [random() for i in range(1000)]` using a traditional for loop, you would first initialize an empty list and then append a random number to this list in each iteration of the loop. Here's how you can do it:

[23]:
```python
from random import random

l1 = []  # Initialize an empty list
for i in range(1000):
    l1.append(random())  # Append a random number to the list in each iteration
```

The output tells you that the code was very quick to execute, with an average time of 26.8 microseconds per loop, and the execution time was quite consistent, with a standard deviation of about 828 nanoseconds. The measurement was based on a total of 70,000 executions of the loop (10,000 loops per run over 7 runs).

[6]:
```r
%%R

library(microbenchmark)

# Generate random numbers
l1 <- runif(1000)
l2 <- runif(1000)

# Time the operation
timing_result <- microbenchmark(
  s <- sum(l1 / l2),
  times = 100L  # Number of times to run the expression
)

print(timing_result)
```

```
Unit: microseconds
            expr   min     lq    mean median     uq    max neval
 s <- sum(l1/l2) 2.764 2.8045 3.20613 2.8495 2.9435 17.567   100
```

The R results show that the operation is very quick, with most executions taking between 3.4 to 5.8 microseconds, but with some variability as indicated by the range from the minimum to the maximum times. The mean and median are close in value, suggesting a relatively symmetric distribution of execution times.

`l1 <- runif(1000)` in R is more concise compared to the equivalent Python code `l1 = [random() for i in range(1000)]`. This difference in conciseness can often be seen between R and Python, especially in tasks related to data generation and statistical operations.

R is a language designed specifically for statistical computing and has many built-in functions that are highly optimized for such tasks. Functions like `runif` provide a very straightforward and concise way to generate random numbers, which is a common requirement in statistical analysis and simulation.

Python, while extremely versatile and powerful, is a general-purpose programming language. Operations like generating random numbers require either using list comprehensions (as in your example) or loops, along with the standard library functions like random.random(). Python's approach is more verbose but also very flexible, allowing for complex operations within list comprehensions and loops.

Each language has its strengths, and in this case, R's specialized nature for statistical tasks makes it more concise for generating a sequence of random numbers.

Now, let's transform the two lists into NumPy tables with the `array()` method, and do the same calculation with a NumPy method:

[8]:
```
a1 = np.array(l1)
a2 = np.array(l2)
%timeit s = np.sum(a1/a2)
```

```
5.6 µs ± 36 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

As can be seen by executing these codes in an IPython environment, the execution time is much faster with the NumPy methods for this calculation. The creation of an array can be done with the `array()` method, from a list, as we just did:

[13]:
```
list = [1,2,4]
table = np.array(list)
print(table)

# OR

table2 = np.array([1,2,4])
print(table2)

# Or

list_2 = [ [1,2,3], [4,5,6] ]
table_2 = np.array(list_2)
print(table_2)

print(type(table_2))
```

```
[1 2 4]
[1 2 4]
[[1 2 3]
```

 ```
 [4 5 6]]
<class 'numpy.ndarray'>
```

Arrays in NumPy (Numerical Python) are commonly used for creating and working with matrices in Python. NumPy provides a powerful array object, which is a multi-dimensional (n-dimensional) homogeneous array. It's a central data structure of the NumPy library.

NumPy arrays can be used to represent both vectors (1D arrays) and matrices (2D arrays), as well as higher-dimensional data structures. For matrix operations, 2D arrays are typically used. Here's a basic example of how you can create and manipulate matrices with NumPy:

[14]:
```python
# Create a 2x3 matrix
matrix = np.array([[1, 2, 3], [4, 5, 6]])
print(matrix)

# Perform matrix operations
transpose = matrix.T  # Transpose of the matrix
print("Transpose:\n", transpose)

product = np.dot(matrix, transpose)  # Matrix multiplication
print("Matrix Product:\n", product)
```

```
[[1 2 3]
 [4 5 6]]
Transpose:
 [[1 4]
 [2 5]
 [3 6]]
Matrix Product:
 [[14 32]
 [32 77]]
```

10.5 Some Functions Generating array Objects

[20]:
```python
x = np.zeros(4)
print(x)

#The type of zeros (e. g. int, int32, int64, int64, float, float32, float64,␣
↪etc.)
#can be specified using the dtype argument:

print(x.dtype)

x = x.astype("int")
print(x.dtype)

y = np.zeros((4,3))
print(y)
```

```
z = np.zeros((2,4,3))
print(z)
```

```
[0. 0. 0. 0.]
float64
int64
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
[[[0. 0. 0.]
  [0. 0. 0.]
  [0. 0. 0.]
  [0. 0. 0.]]

 [[0. 0. 0.]
  [0. 0. 0.]
  [0. 0. 0.]
  [0. 0. 0.]]]
```

[15]:
```
x = list(range(1,11))
print(x)

y = np.array(x).reshape(2,5)
print(y)

# Two ways to use shape
print(y.shape)
np.shape(y)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]]
(2, 5)
```

[15]: (2, 5)

[33]:
```
shape = np.shape(y)
# Number of rows
n_rows = shape[0]
print("Number of rows:", n_rows)

# Number of columns
n_cols = shape[1]
print("Number of columns:", n_cols)

print(y.shape[0], y.shape[1])
```

```
Number of rows: 2
Number of columns: 5
2 5
```

R is particularly well-suited for statistical and matrix operations, and as such, it often provides more straightforward ways to perform these kinds of tasks. The matrix function in R is a great example of this. R's matrix function directly takes the data (or, the vector x) and the desired dimensions (number of rows and columns) as arguments, making it very concise and easy to use for creating matrices.

This ease of use for matrix and array operations is one of the reasons why R is so popular in data analysis, statistics, and related fields. Python also has powerful capabilities for these tasks, especially with libraries like NumPy, but the syntax can be more verbose compared to R for certain operations.

[12]:
```
%%R
y <- matrix(1:10, 2, 5)
print(y)
dim(y)
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
[1] 2 5
```

10.6 Extracting Elements from an Array

Access to the elements of an array is done in the same way as for lists, using indexes. The syntax is as follows: `array[lower:upper:step]` - When `lower` is not specified, the first element (indexed 0) is considered as the value assigned to lower. - When `upper` is not specified, the last element is considered as the value assigned toupper'. - When `step` is not specified, a step of 1 is assigned by default.

[21]:
```
print(y)
y[1,3]
```

```
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]]
```

[21]: 9

[22]:
```
y[,3]
```

```
  Cell In[22], line 1
    y[,3]
      ^
SyntaxError: invalid syntax
```

[23]:
```
y[:,3]
```

[23]:
```
array([4, 9])
```

[27]:
```
y[0,:]
```

[27]:
```
array([1, 2, 3, 4, 5])
```

[28]:
```
y[0,:-1]
```

[28]:
```
array([1, 2, 3, 4])
```

[36]:
```
y[0,2:4]
```

[36]:
```
array([3, 4])
```

[29]:
```
len(y[0,:-1])
```

[29]:
```
4
```

[38]:
```
A = np.array(np.arange(16)).reshape((4, 4))
A
```

[38]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

[37]:
```
A[1,2]
```

[37]:
```
6
```

To select multiple rows at a time, we can pass in a list specifying our selection. For instance, `[1,2]` will retrieve the second and third rows:

[39]:
```
A[[1,2]]
```

[39]:
```
array([[ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

To select the first and third columns, we pass in `[0,2]` as the second argument in the square brackets. In this case we need to supply the first argument `:` which selects all rows.

[40]:
```
A[:,[0,2]]
```

[40]:
```
array([[ 0,  2],
       [ 4,  6],
       [ 8, 10],
```

```
        [12, 14]])
```

Now, suppose that we want to select the submatrix made up of the second and fourth rows as well as the first and third columns. This is where indexing gets slightly tricky. It is natural to try to use lists to retrieve the rows and columns:

[41]:
```
A[[1,3],[0,2]]
```

[41]:
```
array([ 4, 14])
```

When supplied with two indexing lists, the numpy interpretation is that these provide pairs of i, j indices for a series of entries. That is why the pair of lists must have the same length. However, that was not our intent, since we are looking for a submatrix. One easy way to do this is as follows. We first create a submatrix by subsetting the rows of `A`, and then on the fly we make a further submatrix by subsetting its columns.

[42]:
```
A[[1,3]][:,[0,2]]
```

[42]:
```
array([[ 4,  6],
       [12, 14]])
```

An easier way:

[43]:
```
idx = np.ix_([1,3],[0,2,3])
A[idx]
```

[43]:
```
array([[ 4,  6,  7],
       [12, 14, 15]])
```

The convenience function `np.ix_()` allows us to extract a submatrix using lists, by creating an intermediate mesh object.

Here, the function `np.all()` has checked whether all entries of an array are `True`. A similar function, `np.any()`, can be used to check whether any entries of an array are `True`.

[44]:
```
np.all(A == 0)
```

[44]:
```
False
```

[45]:
```
np.any(A == 6)
```

[45]:
```
True
```

10.7 Missing Values

In Python, especially when working with data science and machine learning, missing values are commonly represented using `None` or `NaN` (Not a Number). The choice between these two depends on the context and the type of data you are dealing with. `None`: This is the Python equivalent of `NULL` in other languages. It's a Python object that often represents the absence of a value. None is used in general Python programming to denote the absence of a value or a null state. `NaN`: This is a special floating-point value defined in the IEEE floating-point standard. It is used to represent missing or undefined numerical data. In Python, NaN is used primarily in numerical arrays and is part of the NumPy library, which is extensively used in data science and machine learning.

When working with pandas, a popular data manipulation library in Python, both `None` and `NaN` can represent missing data. Pandas treats `None` and `NaN` as essentially interchangeable for indicating missing or null values. To facilitate this convention, there are several methods for detecting, removing, and replacing null values in pandas

In NumPy, which is a core library for numerical computing in Python, missing values are typically represented using `numpy.nan` for floating-point data. Unlike Python's general-purpose `None`, which can be used with any data type, `numpy.nan` is specifically a floating-point value and fits naturally into arrays of floats.

[47]:
```
a = np.array([1.0, 2.0, NaN, 4.0])
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[47], line 1
----> 1 a = np.array([1.0, 2.0, NaN, 4.0])

NameError: name 'NaN' is not defined
```

[57]:
```
a = np.array([1.0, 2.0, None, 4.0])
a
```

[57]:
```
array([1.0, 2.0, None, 4.0], dtype=object)
```

[60]:
```
b = np.array([1.0, 2.0, np.nan, 4.0])
b
```

[60]:
```
array([ 1.,  2., nan,  4.])
```

[51]:
```
np.isnan(a)
```

[51]:
```
array([False, False,  True, False])
```

[61]:
```
np.isnan(b)
```

[61]:
```
array([False, False,  True, False])
```

[52]:
```
np.isnan(a).any()
```

[52]:
```
True
```

[53]:
```
# Calculate the mean, ignoring NaN values
np.nanmean(a)
```

[53]: 2.3333333333333335

[62]:
```
np.nanmean(b)
```

[62]: 2.3333333333333335

[54]:
```
# Example arrays
b = np.array([10, 20, 30, 40])
c = np.array([2, 0, 5, 0])

d = np.divide(b, c)

print(d)
```

```
[ 5. inf  6. inf]
```

```
/var/folders/wt/4xtk6v051vd349k3wktfld480000gn/T/ipykernel_55077/1948954575.py:6
: RuntimeWarning: divide by zero encountered in divide
  d = np.divide(b, c)
```

[56]:
```
# Safe division
d = np.divide(b, c, where=c!=0)

print(d)
```

```
[ 5. -0.  6. -0.]
```

[63]:
```
import pandas as pd

# Create a DataFrame with missing values
# Will See Pandas in CH.10 Below
df = pd.DataFrame({'A': [1, 2, None], 'B': [4, None, 6], 'C': [7, 8, np.nan]})

# Output the DataFrame
print(df)
```

```
     A    B    C
0  1.0  4.0  7.0
1  2.0  NaN  8.0
2  NaN  6.0  NaN
```

10.8 Conditional indexing

Conditional indexing in Python, especially with NumPy arrays and Pandas DataFrames, is a powerful feature that allows you to select elements based on some condition. Let's go through examples for both NumPy arrays and Pandas DataFrames:

[41]:
```
import numpy as np

# Creating a NumPy array
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

# Condition: Select elements greater than 5
selected_elements = arr[arr > 5]
print(selected_elements)

# Creating a 2D NumPy array (matrix)
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# Condition: Select elements greater than 5
selected_elements = matrix[matrix > 5]
print(selected_elements)

matrix[matrix > 5]
```

```
[ 6  7  8  9 10]
[6 7 8 9]
```

[41]:
```
False
```

[43]:
```
# Condition: Find indices of elements greater than 5
indices = np.where(matrix > 5)

print(indices)
print(indices[0])
print(indices[1])
```

```
(array([1, 2, 2, 2]), array([2, 0, 1, 2]))
[1 2 2 2]
[2 0 1 2]
```

[44]:
```
# Using indices to extract the values from the matrix
selected_values = matrix[indices]

print(selected_values)
```

```
[6 7 8 9]
```

[37]:
```
import pandas as pd

# Creating a DataFrame
df = pd.DataFrame({
    'A': [1, 2, 3, 4, 5],
```

```
    'B': [5, 4, 3, 2, 1],
    'C': [2, 3, 4, 5, 6]
})

# Condition: Select rows where column 'A' is greater than 2
selected_rows = df[df['A'] > 2]
print(selected_rows)

# Condition: Select rows where 'A' is greater than 2 and 'B' is less than 5
selected_rows = df[(df['A'] > 2) & (df['B'] < 5)]
print(selected_rows)
```

```
   A  B  C
2  3  3  4
3  4  2  5
4  5  1  6
   A  B  C
2  3  3  4
3  4  2  5
4  5  1  6
```

[49]:
```
%%R

X <- matrix(1:10, 2, 5)
print(X[X>5])

# or

ind <- which(X>5, arr.ind = TRUE)
print(ind)

print(X[ind])
```

```
[1]  6  7  8  9 10
     row col
[1,]   2   3
[2,]   1   4
[3,]   2   4
[4,]   1   5
[5,]   2   5
[1]  6  7  8  9 10
```

[53]:
```
%%R

# Creating a DataFrame
df = data.frame(A = 1:5,
                B = 5:1,
```

```
              C = 2:6)

# Condition: Select rows where column 'A' is greater than 2
selected_rows = df[df$A > 2, ]
print(selected_rows)

# Condition: Select rows where 'A' is greater than 2 and 'B' is less than 5
selected_rows = df[df$A > 2 & df$B < 5, ]
print(selected_rows)

# which()

ind <- which(df$A > 2 & df$B < 5)
selected_rows = df[ind, ]
print(selected_rows)
```

```
  A B C
3 3 3 4
4 4 2 5
5 5 1 6
  A B C
3 3 3 4
4 4 2 5
5 5 1 6
  A B C
3 3 3 4
4 4 2 5
5 5 1 6
```

R has a reputation for being particularly concise and straightforward for certain types of data operations, especially those involving matrices and data frames. This is largely because R is specifically designed for statistical computing and data analysis, so operations like matrix manipulation are very streamlined.

In R, functions like `rbind()` and `cbind()` are directly tailored for adding rows and columns to matrices, making the syntax very clean and intuitive for these operations. This focus on data-centric tasks is one of the strengths of R and a reason why it's so popular in the statistics and data science communities.

Python, being a more general-purpose language, can require more verbose code for the same operations, particularly when using libraries like NumPy for numerical computing. However, Python's broad applicability across different domains, from web development to machine learning, makes it a versatile choice.

Both languages have their strengths, and the choice often depends on the specific needs and context of the task at hand. For pure data analysis and statistical work, R might be more convenient, while Python offers greater flexibility for a wide range of applications.

[54]:
```
%%R

mat <- matrix(1:9, 3, 3)
r <- c(1,1,-5)
co <- c(19, 0.57, 22, 42)

mat1 <- rbind(mat, r)
mat2 <- cbind(mat1, co)
mat2
```

```
           co
  1 4  7 19.00
  2 5  8  0.57
  3 6  9 22.00
r 1 1 -5 42.00
```

[2]:
```python
import numpy as np

# Create a 3x3 matrix
mat = np.array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])

# New row and column to be added
r = np.array([1, 1, -5])
co = np.array([19, 0.57, 22, 42])

# Add the new row to the bottom of the matrix
mat1 = np.vstack([mat, r])

# Add the new column to the right of the matrix
# First, reshape 'co' to be a 4x1 column vector
co = co.reshape(-1, 1)  # '-1' lets NumPy automatically calculate the size
mat2 = np.hstack([mat1, co])

print("Original Matrix:\n", mat)
print("Matrix with New Row:\n", mat1)
print("Matrix with New Row and Column:\n", mat2)
```

```
Original Matrix:
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
Matrix with New Row:
 [[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [ 1  1 -5]]
```

```
Matrix with New Row and Column:
 [[ 1.     2.     3.    19.   ]
 [ 4.     5.     6.     0.57]
 [ 7.     8.     9.    22.   ]
 [ 1.     1.    -5.    42.   ]]
```

In Python, the append method for lists and the `numpy.append` function for NumPy arrays work differently than the `vstack` and `hstack` functions we used above. Let's clarify the differences:

List append Method in Python: - The append method is used with Python lists to add a single element to the end of the list. - It does not return a new list; rather, it modifies the existing list in place. - It cannot be used directly for adding a row or a column to a 2D list or a NumPy array in the way `vstack` or `hstack` do.

[ ]:
```
my_list = [1, 2, 3]
my_list.append(4)  # Adds 4 to the end of my_list
print(my_list)  # Output: [1, 2, 3, 4]
```

NumPy append Function: - `numpy.append` adds values to the end of an array. - Unlike list append, it returns a new array and does not modify the original array in place. - It flattens the array by default if you don't specify the axis, which means it will return a 1D array even if you append a row or a column to a 2D array. - To use it for adding a row or a column without flattening, you need to specify the axis parameter.

[58]:
```
mat = np.array([[1, 2, 3], [4, 5, 6]])
new_row = np.array([7, 8, 9])

# Appending a new row without flattening
mat_with_row = np.append(mat, [new_row], axis=0)
print(mat_with_row)

new_row = np.array([7, 8, 9])

# Appending a new column without flattening
new_col = np.array([7, 8])
mat_with_col = np.append(mat, [new_col], axis=1)
print(mat_with_col)
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[58], line 12
     10 # Appending a new column without flattening
     11 new_col = np.array([7, 8])
---> 12 mat_with_col = np.append(mat, [new_col], axis=1)
     13 print(mat_with_col)
```

```
File <__array_function__ internals>:180, in append(*args, **kwargs)

File ~/anaconda3/lib/python3.10/site-packages/numpy/lib/function_base.py:5444,␣
↪in append(arr, values, axis)
   5442     values = ravel(values)
   5443     axis = arr.ndim-1
-> 5444 return concatenate((arr, values), axis=axis)

File <__array_function__ internals>:180, in concatenate(*args, **kwargs)

ValueError: all the input array dimensions for the concatenation axis must matc␣
↪exactly, but along dimension 0, the array at index 0 has size 2 and the array␣
↪at index 1 has size 1
```

For column-wise appending (along `axis=1`), as in appending `new_col` to `mat`, you often need to reshape the array to match the number of rows in `mat`, especially if `new_col` starts as a 1D array. But for row-wise appending (along `axis=0`), as long as the number of elements in the row being appended matches the number of columns in the matrix, additional reshaping isn't necessary.

Also note that, in our row-wise appending above, when appending `new_row` to mat using `np.append`, the square brackets `[new_row]` are used to ensure that `new_row` is treated as a two-dimensional array with one row, rather than as a one-dimensional array. This is important for aligning the dimensions correctly when appending along `axis=0` (the row axis).

[59]:
```
new_col = np.array([7, 8])
mat_with_col = np.append(mat, new_col.reshape(2, 1), axis=1)
print(mat_with_col)
```

```
[[1 2 3 7]
 [4 5 6 8]]
```

10.9 Squences and Slice Notations

The function `np.linspace()` can be used to create a sequence of numbers.

[32]:
```
seq1 = np.linspace(0, 10, 11)
seq1
```

[32]:
```
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.])
```

The function `np.arange()` returns a sequence of numbers spaced out by `step`. If `step` is not specified, then a default value of 1 is used. Let's create a sequence that starts at 0 and ends at 10.

[33]:
```
seq2 = np.arange(0,10)
seq2
```

[33]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Why isn't 10 output above? This has to do with slice notation in Python. Slice notation is used to index sequences such as lists, tuples and arrays. Suppose we want to retrieve the fourth through sixth (inclusive) entries of a string. We obtain a slice of the string using the indexing notation `[3:6]`

```
[34]: "hello world"[3:6]
```

```
[34]: 'lo '
```

In the code block above, the notation `3:6` is shorthand for `slice(3,6)` when used inside `[]`.

```
[35]: "hello world"[slice(3,6)]
```

```
[35]: 'lo '
```

You might have expected `slice(3,6)` to output the fourth through seventh characters in the text string (recalling that Python begins its indexing at zero), but instead it output the fourth through sixth. This also explains why the earlier `np.arange(0, 10)` command output only the integers from 0 to 9. See the documentation slice? for useful options in creating slices

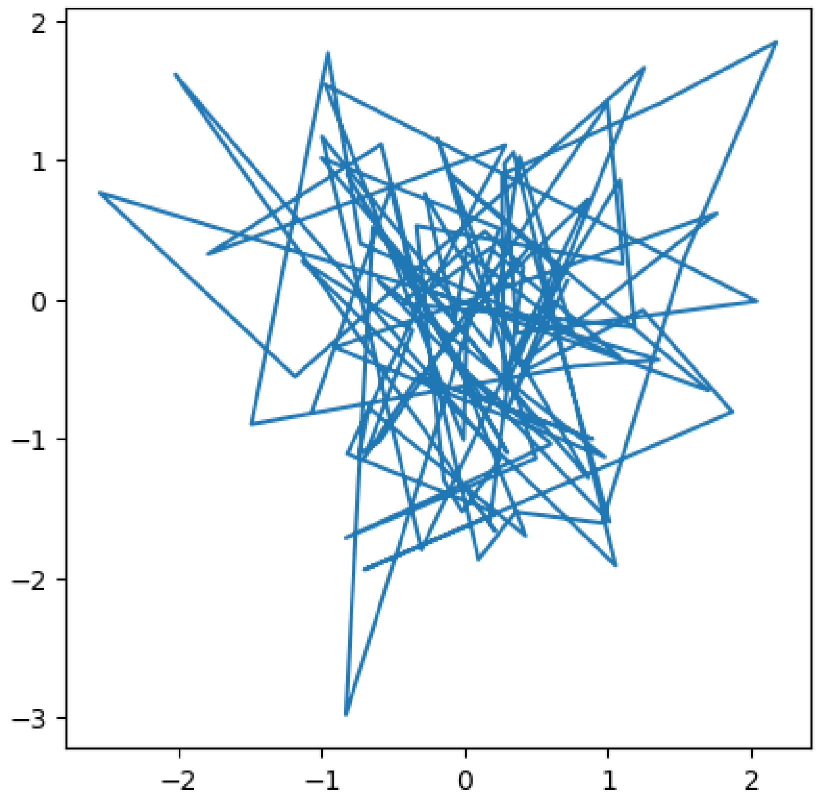
```
[36]: A = np.array(np.arange(16)).reshape((4, 4))
      A
```

```
[36]: array([[ 0,  1,  2,  3],
             [ 4,  5,  6,  7],
             [ 8,  9, 10, 11],
             [12, 13, 14, 15]])
```

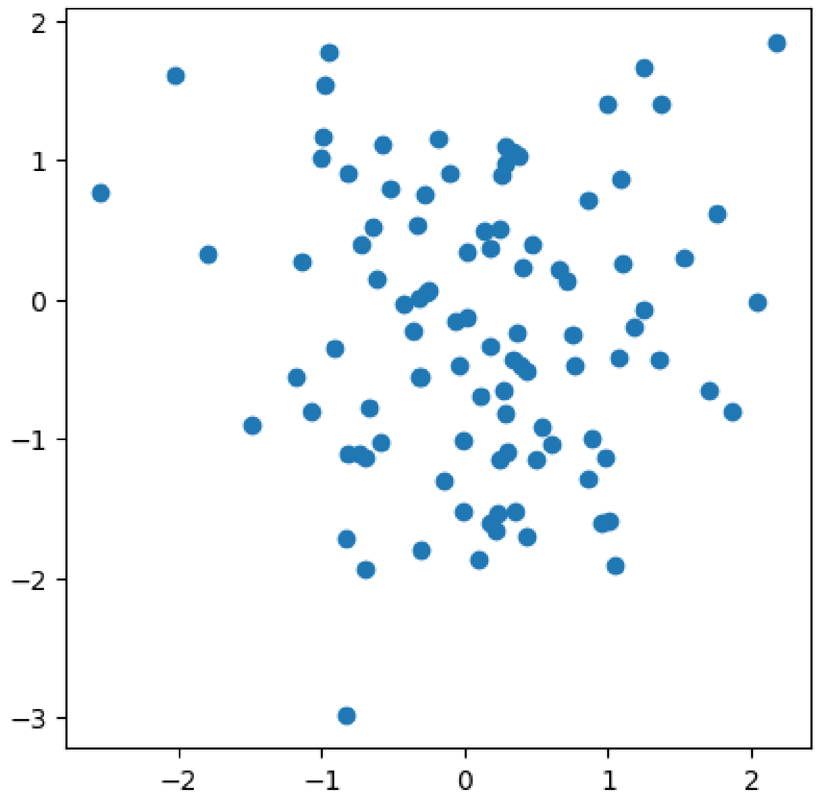
11 Graphics

In Python, common practice is to use the library `matplotlib` for graphics. However, since Python was not written with data analysis in mind, the no- tion of plotting is not intrinsic to the language. We will use the `subplots()` function from `matplotlib.pyplot` to create a figure and the axes onto which we plot our data. For many more examples of how to make plots in Python, readers are encouraged to visit matplotlib.org/stable/gallery/.

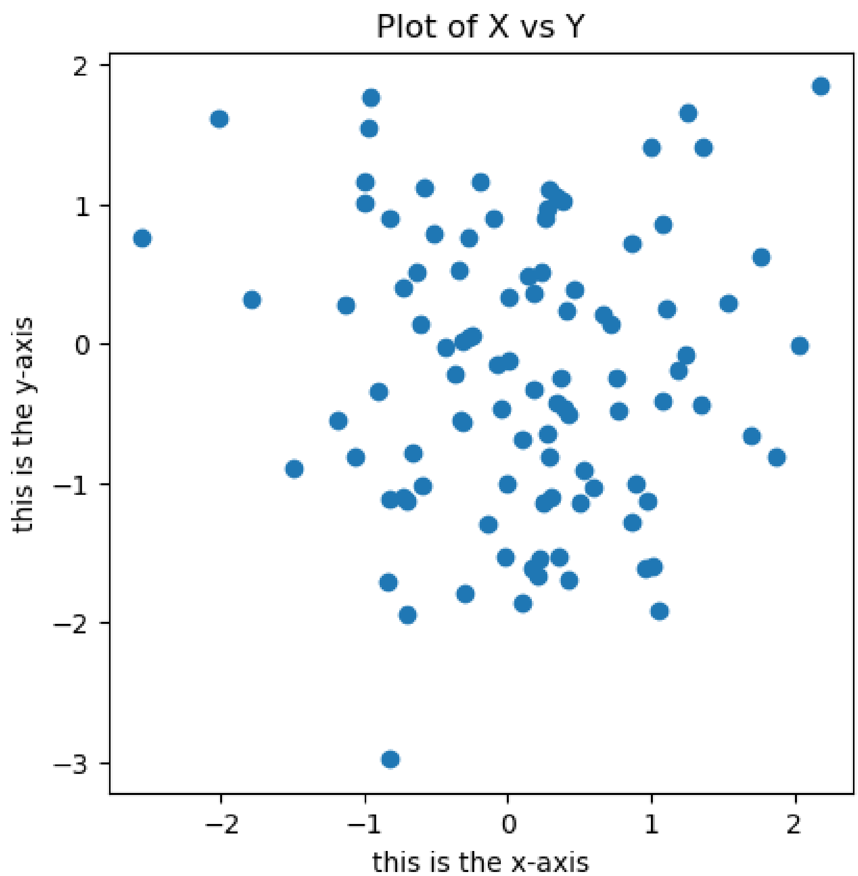
```
[5]: from matplotlib.pyplot import subplots

     # Create a random number generator instance
     rng = np.random.default_rng()

     fig, ax = subplots(figsize=(5, 5))
     x = rng.standard_normal(100)
     y = rng.standard_normal(100)
     ax.plot(x, y)
```



[6]:
```
fig, ax = subplots(figsize=(5, 5))
ax.plot(x, y, 'o')
```



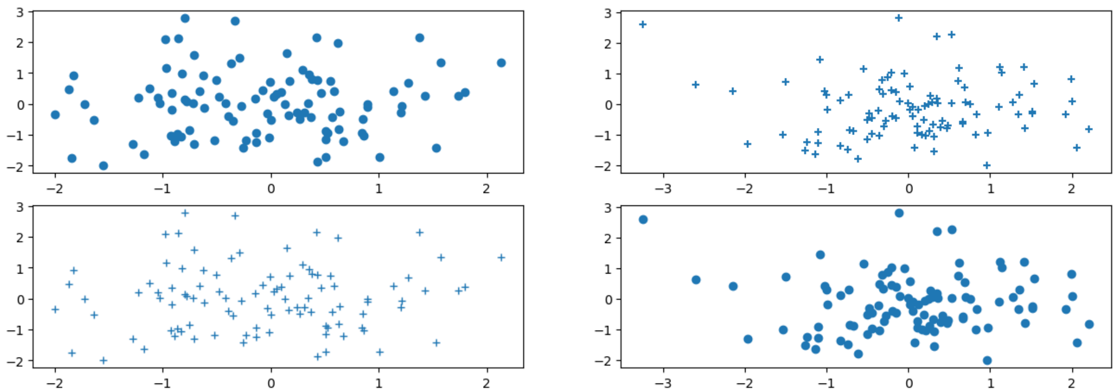
[8]:
```
fig, ax = subplots(figsize=(5, 5))
ax.scatter(x, y, marker='o')
ax.set_xlabel("this is the x-axis")
ax.set_ylabel("this is the y-axis")
ax.set_title("Plot of X vs Y");
```



[27]:
```
import matplotlib.pyplot as plt

# Clear any existing plots
plt.clf()

# Create a random number generator instance
rng = np.random.default_rng()

# Generate random data
x = rng.standard_normal(100)
y = rng.standard_normal(100)
z = rng.standard_normal(100)
w = rng.standard_normal(100)

# Create a 2x2 grid of subplots
fig, axes = subplots(nrows=2, ncols=2, figsize=(15, 5))

# Plot in the first subplot
axes[0,0].plot(x, y, 'o')
```

```
axes[1,0].plot(x, y, '+')
# Scatter plot in the second subplot
axes[0,1].scatter(z, w, marker='+')
axes[1,1].scatter(z, w, marker='o')
plt.show()
```

```
<Figure size 640x480 with 0 Axes>
```



12 Pandas

Data sets often contain different types of data, and may have names as- sociated with the rows or columns. For these reasons, they typically are best accommodated using a data frame. We can think of a data frame as a sequence of arrays of identical length; these are the columns. Entries in the different arrays can be combined to form a row. The pandas library can be used to create and work with data frame objects.

Reading in a Data Set The first step of most analyses involves importing a data set into Python. Before attempting to load a data set, we must make sure that Python knows where to find the file containing it. If the file is in the same location as this notebook file, then we are all set. Otherwise, the command `os.chdir()` can be used to change directory. (You will need to call `import os` before calling `os.chdir()`.) We will begin by reading in `Auto.csv`. This is a comma-separated file, and can be read in using `pd.read_csv()`:

[3]:
```
import numpy as np
import pandas as pd
Auto = pd.read_csv("Auto.csv")
Auto
```

[3]:
```
    mpg  cylinders  displacement horsepower  weight  acceleration  year  \
0  18.0          8         307.0        130    3504          12.0    70
1  15.0          8         350.0        165    3693          11.5    70
2  18.0          8         318.0        150    3436          11.0    70
```

```
3    16.0          8          304.0        150    3433           12.0    70
4    17.0          8          302.0        140    3449           10.5    70
..    …        …         …        …      …          …    …
392  27.0          4          140.0         86    2790           15.6    82
393  44.0          4           97.0         52    2130           24.6    82
394  32.0          4          135.0         84    2295           11.6    82
395  28.0          4          120.0         79    2625           18.6    82
396  31.0          4          119.0         82    2720           19.4    82

     origin                       name
0         1  chevrolet chevelle malibu
1         1          buick skylark 320
2         1         plymouth satellite
3         1              amc rebel sst
4         1                ford torino
..      …                        …
392       1            ford mustang gl
393       2                  vw pickup
394       1              dodge rampage
395       1                ford ranger
396       1                 chevy s-10

[397 rows x 9 columns]
```

[2]:
```
Auto['horsepower']
```

[2]:
```
0      130
1      165
2      150
3      150
4      140
      …
392     86
393     52
394     84
395     79
396     82
Name: horsepower, Length: 397, dtype: object
```

[5]:
```
np.unique(Auto['horsepower'])
```

[5]:
```
array(['100', '102', '103', '105', '107', '108', '110', '112', '113',
       '115', '116', '120', '122', '125', '129', '130', '132', '133',
       '135', '137', '138', '139', '140', '142', '145', '148', '149',
       '150', '152', '153', '155', '158', '160', '165', '167', '170',
       '175', '180', '190', '193', '198', '200', '208', '210', '215',
       '220', '225', '230', '46', '48', '49', '52', '53', '54', '58',
```

```
       '60', '61', '62', '63', '64', '65', '66', '67', '68', '69', '70',
       '71', '72', '74', '75', '76', '77', '78', '79', '80', '81', '82',
       '83', '84', '85', '86', '87', '88', '89', '90', '91', '92', '93',
       '94', '95', '96', '97', '98', '?'], dtype=object)
```

We see the value `?` is being used to encode missing values.

[6]:
```python
Auto = pd.read_csv("Auto.csv",
                  na_values = ["?"])
np.unique(Auto['horsepower'])
```

[6]:
```
array([ 46.,  48.,  49.,  52.,  53.,  54.,  58.,  60.,  61.,  62.,  63.,
        64.,  65.,  66.,  67.,  68.,  69.,  70.,  71.,  72.,  74.,  75.,
        76.,  77.,  78.,  79.,  80.,  81.,  82.,  83.,  84.,  85.,  86.,
        87.,  88.,  89.,  90.,  91.,  92.,  93.,  94.,  95.,  96.,  97.,
        98., 100., 102., 103., 105., 107., 108., 110., 112., 113., 115.,
       116., 120., 122., 125., 129., 130., 132., 133., 135., 137., 138.,
       139., 140., 142., 145., 148., 149., 150., 152., 153., 155., 158.,
       160., 165., 167., 170., 175., 180., 190., 193., 198., 200., 208.,
       210., 215., 220., 225., 230.,  nan])
```

[7]:
```python
Auto['horsepower'].sum()
```

[7]:
```
40952.0
```

[33]:
```python
Auto.shape
```

[33]:
```
(397, 9)
```

[35]:
```python
Auto.isna().any(axis=1).sum()
```

[35]:
```
5
```

There are various ways to deal with missing data. In this case, since only five of the rows contain missing observations, we choose to use the `Auto.dropna()` method to simply remove these rows.

[4]:
```python
Auton = Auto.dropna()
Auton.shape
```

[4]:
```
(397, 9)
```

[40]:
```python
Auton.columns
```

[40]:
```
Index(['mpg', 'cylinders', 'displacement', 'horsepower', 'weight',
       'acceleration', 'year', 'origin', 'name'],
      dtype='object')
```

Accessing the rows and columns of a data frame is similar, but not identical, to accessing the rows and columns of an array. Recall that the first argument to the [] method is always applied to the rows of the array. Similarly, passing in a slice to the [] method creates a data frame whose rows are determined by the slice:

[38]:
```
Auton[:3]
```

[38]:
```
    mpg  cylinders  displacement  horsepower  weight  acceleration  year  \
0  18.0          8         307.0       130.0    3504          12.0    70
1  15.0          8         350.0       165.0    3693          11.5    70
2  18.0          8         318.0       150.0    3436          11.0    70

   origin                       name
0       1  chevrolet chevelle malibu
1       1          buick skylark 320
2       1         plymouth satellite
```

[45]:
```
# Conditional Indexing
idx_80 = Auton['weight'] > 5000
Auton[idx_80]
```

[45]:
```
     mpg  cylinders  displacement  horsepower  weight  acceleration  year  \
44  13.0          8         400.0       175.0    5140          12.0    71

    origin                 name
44       1  pontiac safari (sw)
```

If we pass in a list of strings to the [] method, we obtain a data frame containing the corresponding set of columns.

[46]:
```
Auton[['mpg', 'horsepower']]
```

[46]:
```
      mpg  horsepower
0    18.0       130.0
1    15.0       165.0
2    18.0       150.0
3    16.0       150.0
4    17.0       140.0
..    …          …
392  27.0        86.0
393  44.0        52.0
394  32.0        84.0
395  28.0        79.0
396  31.0        82.0

[392 rows x 2 columns]
```

Using index for selection In database systems, an index is a data structure that improves the speed of data retrieval operations on a database table. Indexes are used to quickly locate data without having to search every row in a database table every time a database table is accessed. When you apply a filter or a query on a column that has an index, the database management system can use the index to find rows faster. This is analogous to an index in a book - instead of reading every page, you refer to the index to find the pages you need. For example, if you have an indexed column `employee_id`, and you query for `employee_id = 12345`, the database can use the index to quickly locate this record.

In data analysis tools like Pandas in Python, indexing can also help in faster data retrieval and filtering. Pandas allows you to set indexes on DataFrames for similar purposes.

When we loaded our data frame, the rows are labeled using integers 0 to 396.

[47]:
```
Auton.index
```

[47]:
```
Int64Index([  0,   1,   2,   3,   4,   5,   6,   7,   8,   9,
            …
            387, 388, 389, 390, 391, 392, 393, 394, 395, 396],
           dtype='int64', length=392)
```

[5]:
```
Auto_re = Auton.set_index('name')
Auto_re
```

[5]:
```
                            mpg  cylinders  displacement horsepower  weight  \
name
chevrolet chevelle malibu  18.0          8         307.0        130    3504
buick skylark 320          15.0          8         350.0        165    3693
plymouth satellite         18.0          8         318.0        150    3436
amc rebel sst              16.0          8         304.0        150    3433
ford torino                17.0          8         302.0        140    3449
…                          …          …           …          …       …
ford mustang gl            27.0          4         140.0         86    2790
vw pickup                  44.0          4          97.0         52    2130
dodge rampage              32.0          4         135.0         84    2295
ford ranger                28.0          4         120.0         79    2625
chevy s-10                 31.0          4         119.0         82    2720

                           acceleration  year  origin
name
chevrolet chevelle malibu          12.0    70       1
buick skylark 320                  11.5    70       1
plymouth satellite                 11.0    70       1
amc rebel sst                      12.0    70       1
ford torino                        10.5    70       1
…                                  …   …     …
ford mustang gl                    15.6    82       1
vw pickup                          24.6    82       2
```

```
dodge rampage                        11.6    82        1
ford ranger                          18.6    82        1
chevy s-10                           19.4    82        1

[397 rows x 8 columns]
```

[49]:
```
# We see that the column 'name' is no longer there.
Auto_re.columns
```

[49]:
```
Index(['mpg', 'cylinders', 'displacement', 'horsepower', 'weight',
       'acceleration', 'year', 'origin'],
      dtype='object')
```

Index entries need not be unique: there are several cars in the data frame named `ford galaxie 500`.

[56]:
```
Auto_re.loc['ford galaxie 500', ['mpg', 'origin']]
```

[56]:
```
                   mpg  origin
name
ford galaxie 500  15.0       1
ford galaxie 500  14.0       1
ford galaxie 500  14.0       1
```

Now, we can access rows of the data frame by `name` using the `loc[]` method of `Auto`:

[50]:
```
rows = ['amc rebel sst', 'ford torino']
Auto_re.loc[rows]
```

[50]:
```
                mpg  cylinders  displacement  horsepower  weight  \
name
amc rebel sst  16.0          8         304.0       150.0    3433
ford torino    17.0          8         302.0       140.0    3449

               acceleration  year  origin
name
amc rebel sst          12.0    70       1
ford torino            10.5    70       1
```

[53]:
```
Auto_re.iloc[[3,4]]
```

[53]:
```
                mpg  cylinders  displacement  horsepower  weight  \
name
amc rebel sst  16.0          8         304.0       150.0    3433
ford torino    17.0          8         302.0       140.0    3449

               acceleration  year  origin
name
```

```
amc rebel sst           12.0    70        1
ford torino             10.5    70        1
```

`loc[]` is label-based; it uses the names or labels of rows/columns. `iloc[]` is integer position-based; it uses the numerical positions in the DataFrame, starting from 0. The same results from both methods indicate a match between the label positions and their numerical positions in this particular instance of your DataFrame.

[54]:
```
# We can also use it to retrieve the 1st, 3rd and and 4th columns
Auto_re.iloc[:,[0,2,3]]
```

[54]:
```
                            mpg  displacement  horsepower
name
chevrolet chevelle malibu  18.0         307.0       130.0
buick skylark 320          15.0         350.0       165.0
plymouth satellite         18.0         318.0       150.0
amc rebel sst              16.0         304.0       150.0
ford torino                17.0         302.0       140.0
…                          …            …           …
ford mustang gl            27.0         140.0        86.0
vw pickup                  44.0          97.0        52.0
dodge rampage              32.0         135.0        84.0
ford ranger                28.0         120.0        79.0
chevy s-10                 31.0         119.0        82.0

[392 rows x 3 columns]
```

We can extract the 4th and 5th rows, as well as the 1st, 3rd and 4th columns, using a single call to `iloc[]`:

[55]:
```
Auto_re.iloc[[3,4],[0,2,3]]
```

[55]:
```
                mpg  displacement  horsepower
name
amc rebel sst  16.0         304.0       150.0
ford torino    17.0         302.0       140.0
```

Suppose that we want to create a data frame consisting of the `year` and `origin` of the subset of cars with `weight` greater than 5000

[60]:
```
idx_80 = Auto_re['weight'] > 5000
Auto_re.loc[idx_80, ['weight', 'origin', 'year']]
```

[60]:
```
                     weight  origin  year
name
pontiac safari (sw)    5140       1    71
```

To do this more concisely, we can use an anonymous function called a `lambda`:

[61]:
```python
Auto_re.loc[lambda df: df['weight'] > 5000, ['weight', 'origin', 'year']]
```

[61]:
```
                     weight  origin  year
name
pontiac safari (sw)    5140       1    71
```

As another example of using a `lambda`, suppose that we want all cars built after 1981 that achieve greater than 30 miles per gallon:

[5]:
```python
Auto_re.loc[lambda df: (df['year'] > 81) & (df['mpg'] > 30),
            ['weight', 'origin']
           ]
```

[5]:
```
                                   weight  origin
name
chevrolet cavalier 2-door            2395       1
pontiac j2000 se hatchback           2575       1
volkswagen rabbit l                  1980       2
mazda glc custom l                   2025       3
mazda glc custom                     1970       3
plymouth horizon miser               2125       1
mercury lynx l                       2125       1
nissan stanza xe                     2160       3
honda accord                         2205       3
toyota corolla                       2245       3
honda civic                          1965       3
honda civic (auto)                   1965       3
datsun 310 gx                        1995       3
oldsmobile cutlass ciera (diesel)    3015       1
toyota celica gt                     2665       3
dodge charger 2.2                    2370       1
vw pickup                            2130       2
dodge rampage                        2295       1
chevy s-10                           2720       1
```

As another example, suppose that we want to retrieve all `Ford` and `Datsun` cars with `displacement` less than 80. We check whether each `name` entry contains either the string `ford` or `datsun` using the `str.contains()` method of the index attribute of of the dataframe:

[70]:
```python
Auto_re.loc[lambda df: (df['displacement'] < 80)
                       & (df.index.str.contains('ford')
                       | df.index.str.contains('datsun')),
            ['weight', 'origin']
           ]
```

[70]:
```
             weight  origin
name
datsun 1200    1613       3
```

```
 datsun b210    1950       3
```

To replicate this type of data filtering in R, we can use a combination of base R functions and, optionally, functions from the `dplyr` package, which is part of the `tidyverse` and provides a more intuitive syntax for data manipulation. Indexing doesn't work in R as it works in Python. Here's how you can achieve the same result:

[17]:
```
%%R -i Auton

# Filter for Ford and Datsun cars with displacement < 80
filtered_cars <- Auton[with(Auton, displacement < 80 &
                              (grepl("ford", name, ignore.case = TRUE) |
                               grepl("datsun", name, ignore.case = TRUE))),
                         c("name","weight", "origin")]
filtered_cars
```

```
           name weight origin
54  datsun 1200   1613      3
129 datsun b210   1950      3
```

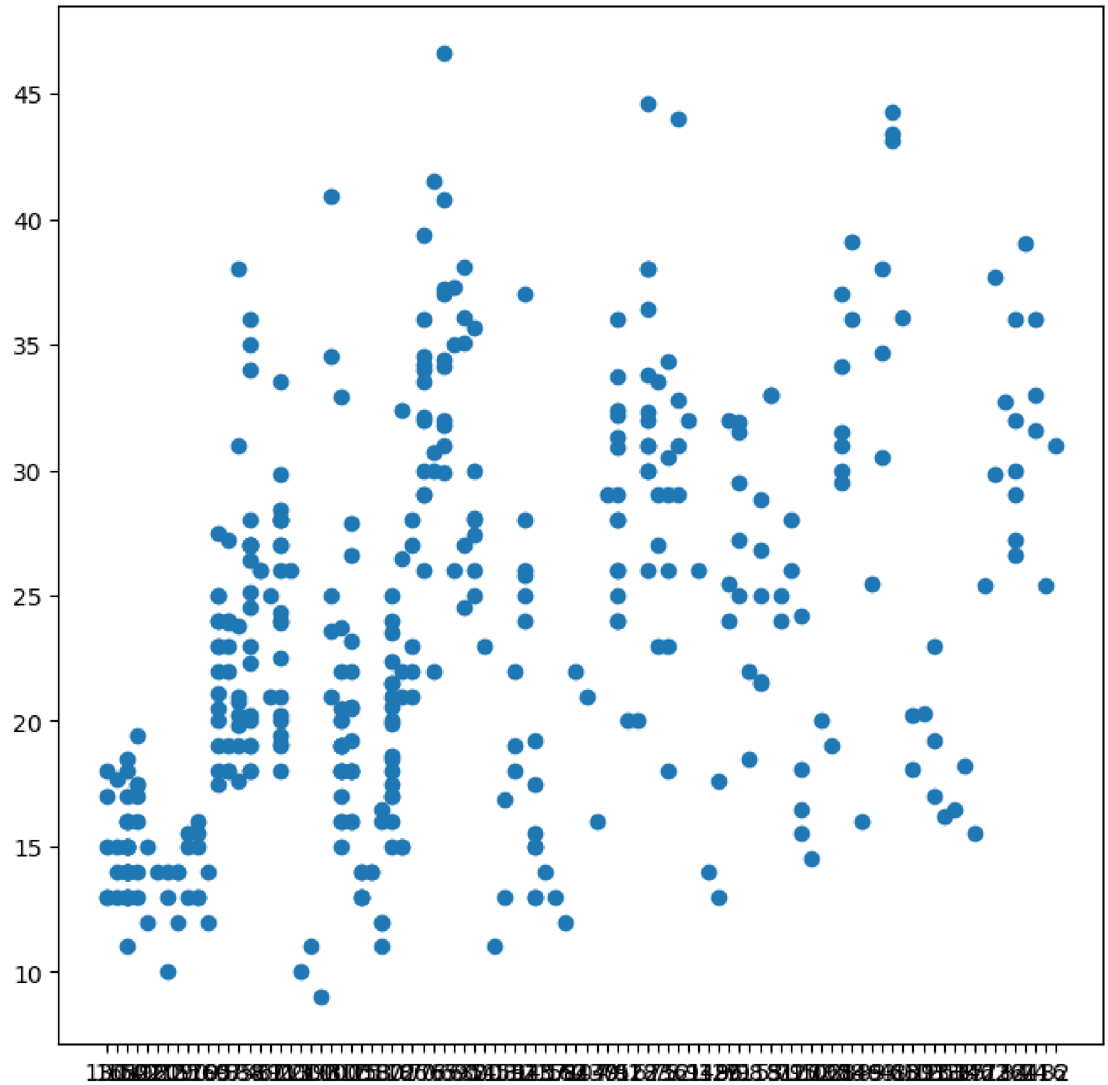
If you're using the `dplyr` package, you can achieve the same with a more readable syntax:

[25]:
```
%%R -i Auton

library(dplyr)
library(tibble)

# Filter using dplyr
fil_cars <- Auton %>%
  filter(displacement < 80,
         grepl("ford", name, ignore.case = TRUE) |
         grepl("datsun", name, ignore.case = TRUE)) %>%
  select(name, weight, origin)

fil_cars
```

```
           name weight origin
54  datsun 1200   1613      3
129 datsun b210   1950      3
```

13 Plots and Numerical Summaries

[38]:
```
from matplotlib.pyplot import subplots

fig, ax = subplots(figsize=(8, 8))
ax.plot(Auto['horsepower'], Auto['mpg'], 'o');
```

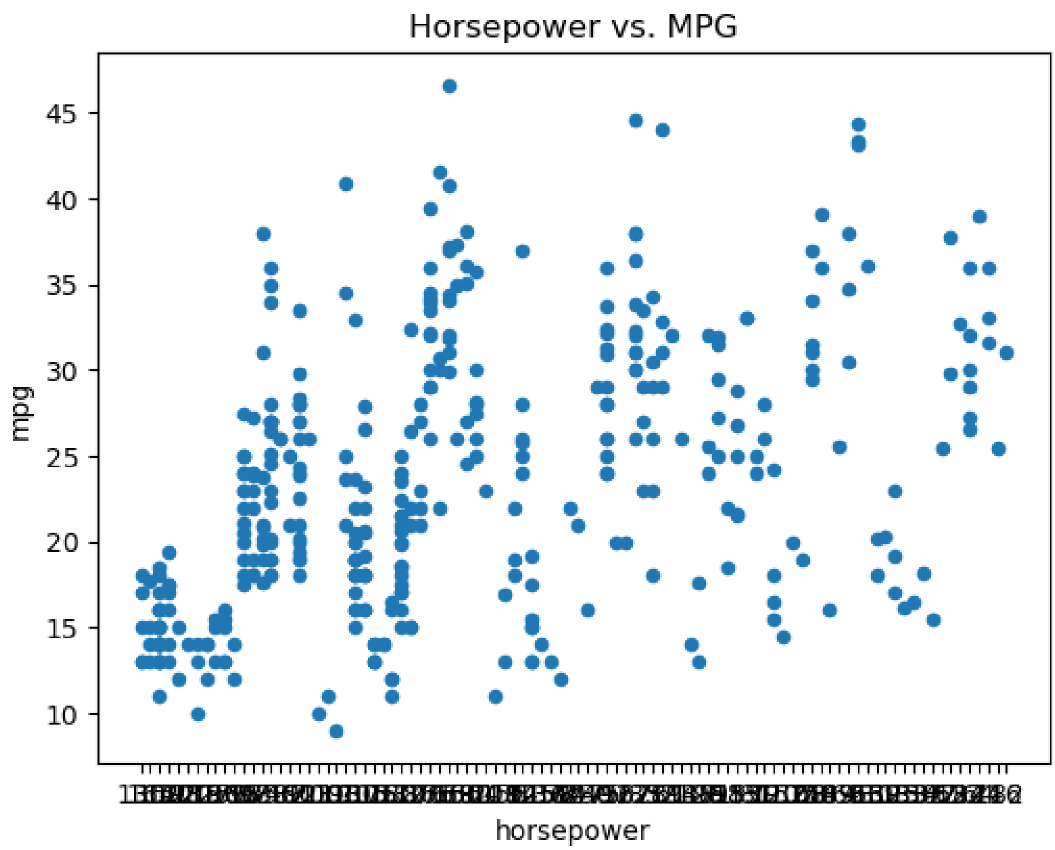


[40]:

```
# Here we use Auto.plot() so the variables can be accessed by
# their names

ax = Auto.plot.scatter('horsepower', 'mpg')
ax.set_title('Horsepower vs. MPG')

# If you want to save
# fig = ax.figure
# fig.savefig("name_of_the figure.png")
```

[40]:

```
Text(0.5, 1.0, 'Horsepower vs. MPG')
```



Consider the `cylinders` variable. Typing in `Auto.cylinders.dtype` reveals that it is being treated as a quantitative variable. However, since there is only a small number of possible values for this variable, we may wish to treat it as qualitative. Below, we replace the `cylinders` column with a categorical version of `Auto.cylinders`. The function `pd.Series()` owes its name to the fact that pandas is often used in time series applications.

[41]:
```
Auto.cylinders.dtype
```

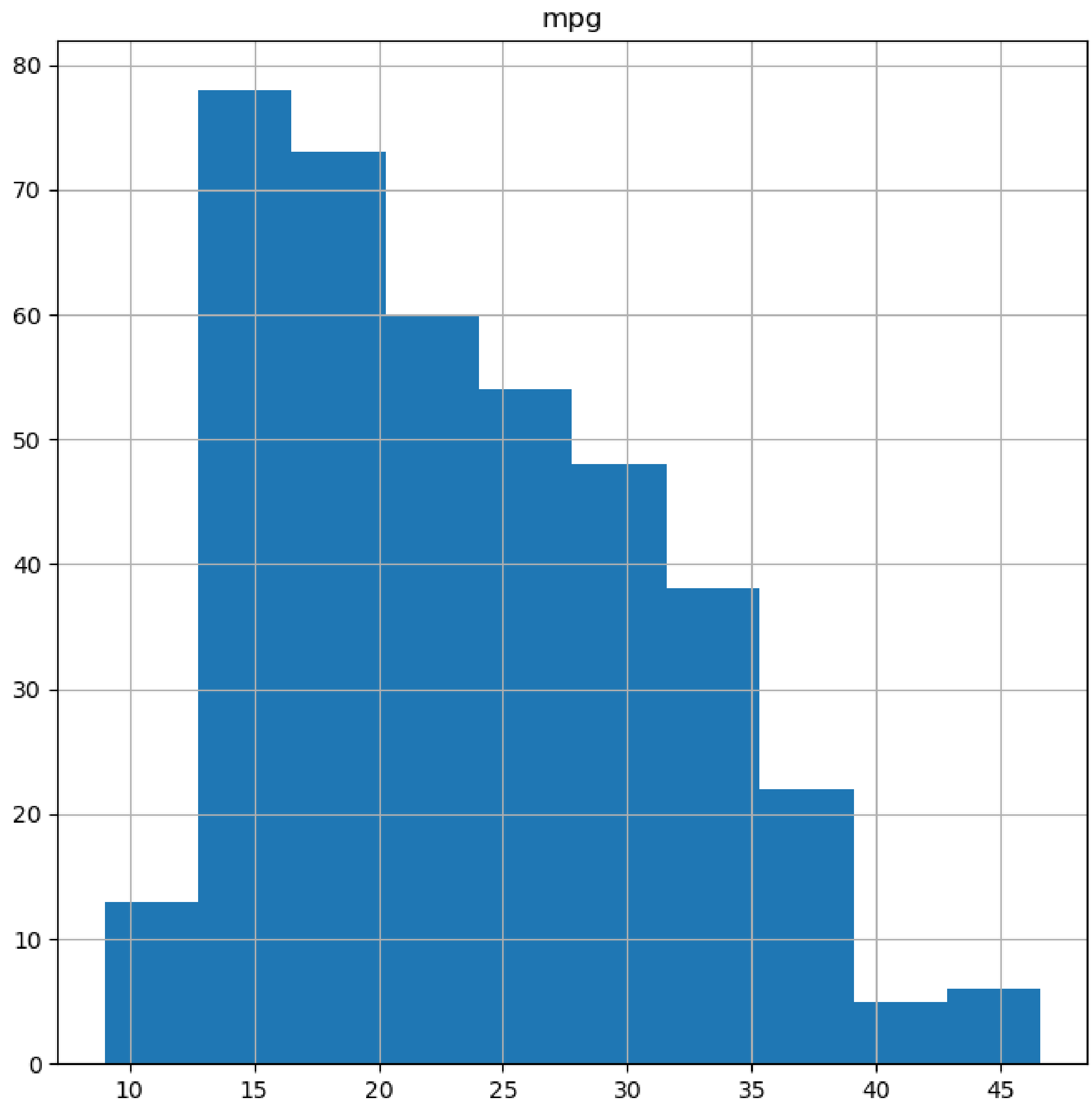
[41]:
```
dtype('int64')
```

[42]:
```
Auto.cylinders = pd.Series(Auto.cylinders, dtype='category')
Auto.cylinders.dtype
```

[42]:
```
CategoricalDtype(categories=[3, 4, 5, 6, 8], ordered=False)
```

Now that cylinders is qualitative, we can display it using the `boxplot()` or the `hist()` method.

[43]:
```
fig, ax = subplots(figsize=(8, 8))
Auto.hist('mpg', ax=ax);
```



The `describe()` method produces a numerical summary of each column in a data frame.

[44]:
```
Auto[['mpg', 'weight']].describe()
```

[44]:

	mpg	weight
count	397.000000	397.000000
mean	23.515869	2970.261965
std	7.825804	847.904119
min	9.000000	1613.000000
25%	17.500000	2223.000000
50%	23.000000	2800.000000
75%	29.000000	3609.000000
max	46.600000	5140.000000

14 Linear Regression

[ ]:
```python
import numpy as np
import pandas as pd
from matplotlib.pyplot import subplots
```

14.1 TBC:-)