Data, Analytics & AI

Python For Data

Python Data Processing

When processing data sets we almost always want to repeat the same operation *for every element* of that dataset..

In [1]:
```
balances = [1_000, 2_500, 10_678, -2_000]
```

In [2]:
```
decisions = [0, 1, 1, 0]  # 0 = No, 1 = Yes
```

We could copy/paste,

In [3]:
```
entry = balances[0]
if entry > 1_500:
    print(entry, "Yes")

entry = balances[1]
if entry > 1_500:
    print(entry, "Yes")


entry = balances[2]
if entry > 1_500:
    print(entry, "Yes")


entry = balances[3]
if entry > 1_500:
    print(entry, "Yes")
```

```
2500 Yes
10678 Yes
```

This approach isnt sustainable, in general, we wont know how many entries in the collection there will be (eg. consider a database, file, etc.).

Python has a *looping* (repeating) syntax which works on datasets,

In [4]:
```
for entry in balances:  # entry = balances[index], REPEAT:
    if entry > 1_500:
        print(entry, "Yes")
```

```
2500 Yes
10678 Yes
```

In [5]:
```
balances
```

Out[5]:
```
[1000, 2500, 10678, -2000]
```

Consider the example below, which repeats `print()` for every entry in `balances`,

In [6]:
```
# b = balances[index]
for b in balances:
    # repeat this code for each entry in balances
    print(b)
```

```
1000
2500
10678
-2000
```

English Algorithms in Python

Suppose, in english, I want to:

```
FOR EACH ENTRY, CALL IT b,
    IN THE DATASET balances
        REPORT THE VALUE of b
```

```
FOR EACH ENTRY, CALL IT balance,
    IN THE DATASET balances
            REPORT THE balance
            AND WHETHER THE balance is MORE THAN 100
```

```
FOR EACH ENTRY, CALL IT customer_balance,
    IN THE DATASET balances
            REPORT WHETHER THE customer_balance is POSITIVE
```

Let's write those in python,

In [7]:
```
for b  in balances:
    print(b)
```

```
1000
2500
10678
-2000
```

In [8]:
```
for balance in balances:
    if balance > 100:
        print(balance, "is more than 100")
```

```
1000 is more than 100
2500 is more than 100
10678 is more than 100
```

In [9]:
```
for customer_balance in balances:
    if customer_balance > 0:
        print(customer_balance, "is positive")
```

```
1000 is positive
2500 is positive
```

```
10678 is positive
```

Understanding the Syntax

```
for TEMPORARY_NAME_OF_ENTRY in EXISTING_DATASET
REPEAT:
        CODE_TO_REPEAT( TEMPORARY_NAME_OF_ENTRY )
```

We can loop over lots of different types of collections,

In [10]:
```
names = ["Alice", "Eve", "Bob"]
```

In [11]:
```
for n in names:
    print(n[0].upper()) # uppercase first letter
```

```
A
E
B
```

In [12]:
```
prices = [2, 54, 7]

for p in prices:
    print(f"The price is £{p}.00")
```

```
The price is £2.00
The price is £54.00
The price is £7.00
```

Exercise

Part 1

Consider the survey below which asks several questions and records their answers...

In [13]:
```
questions = [
    "How happy are you working from home (/5)?",
    "How technical is technical is your job (/5)?",
    "To what degree have you saved money (-5, 5)?",
]
```

In [14]:
```
# answers = []

# for q in questions:
#     answer = input(q)

#     answers.append(answer)
```

In [15]:
```
# answers
```

- Modify the above loop to
  - store the `float()` of your answer
    - HINT: use `float()` on anwers
  - print the answer

- EXTRA: f"" to add a little formatting
- eg., "Your answer was..."

Part 2

- write a loop over the `answers` variable
  - if any answer is more than 3, report "That's Bad!"
  - otherwise, report "That's GOOD!"
- HINT:

```
for .. in ..:
  if ....:
  ...
  else:
  ...
```

Advanced: Built-In Operations for Looping

Python has some built-in operations that help us when looping some specific situations,

- `zip()`
- `range()`
- others
  - `enumerate()`

In [16]:
```
for i in range(0, 10, 2):
    print(i)
```

```
0
2
4
6
8
```

In [17]:
```
numbers = [10, 23, 13]
colors = ["R", "G", "B"]

for n, c in zip(numbers, colors):
    print(n, c)
```

```
10 R
23 G
13 B
```

In [18]:
```
numbers = [10, 23, 13]
colors = ["R", "G", "B"]

for i, n, c in zip(range(0, 3), numbers, colors):
    print(i, n, c)
```

```
0 10 R
1 23 G
2 13 B
```

In [19]:
```
colors = ["R", "G", "B"]
```

```
for i, c in enumerate(colors): # enumerate = range + zip
    print(i, c)
```

```
0 R
1 G
2 B
```

Recap: Why Python?

(because it has lots of useful libraries...)

The `import` keywords loads external "libraries" (systems of tools) that can solve various problems...

very fast numerical programming

In [20]:
```
import numpy
```

very fast spreadsheet/tabular data processing

In [21]:
```
import pandas
```

very fast visualization...

In [22]:
```
import seaborn
```

very fast predictive analysis ("machine learning")

In [23]:
```
import sklearn
```

...some problem-specifc ones...

quite good network analysis

In [24]:
```
import networkx
```

Eg., loading the natural language toolkit...

In [25]:
```
import nltk
```

What are the python data science and analysis libraries?

- NumPy
  - fast lists (aka. arrays)
- Pandas
  - provides single-machine dataframes (aka. tables)
- Seaborn & matplotlib
  - visualization

- Spark
  - query over distributed file systems
- plotly
  - interactive visuals
- scipy, sklearn, tensorflow, pytorch, statsmodels
  - scientific & statistical programming
- Aside:
  - NB. sqlite3 is written in C

Why do we *need* to use them?

Python is very very very slow & memory inefficient

In [26]:
```
from random import random
```

In [27]:
```
dataset = []

for _ in range(1_000_000):
    dataset.append(random())
```

In [28]:
```
%%timeit

total = 0

for x in dataset:
    total += x
```

```
46.5 ms ± 7.66 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

In [29]:
```
import numpy as np
```

In [30]:
```
ds = np.random.uniform(0, 1, 1_000_000)
```

In [31]:
```
%%timeit

ds.sum()
```

```
1.08 ms ± 64.9 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

In [32]:
```
22E-3 / 500E-6
```

Out[32]:
```
44.0
```

Note here the python `for` loop is *44x* slower than the equivalent `.sum()` using `numpy`.

How do these other libraries beat python?

They aren't python... they are written in FORTRAN and C. So when you call `.sum()` you are *NOT* running `python` code, by mostly FORTRAN code.

If you use python keywords (eg., `for`) your program will be very slow. These libraries are not *written in python*, they are compiled from other languages and are "available" from python.

Python is very very very slow & memory inefficient

Python uses 10% more memory:

In [33]:
```
import sys
100 * round(1 -  ds.nbytes / sys.getsizeof(dataset), 1)
```

Out[33]:
```
10.0
```

And `NumPy` is, here, 10x faster... (often much more than this...)

In [34]:
```
#     3.7 ms / 404 µs
round(3.7E-3/404E-6, 1)
```

Out[34]:
```
9.2
```

Python Data Science Libraries

Python data science libraries have conventional *aliases*.

Python data science libraries are extremely fast *additions* to python which add highly efficient data processing tools.

Python itself is very slow (`for`, `if`, etc. are slow! as are lists...). These additions build in much faster data structures & operations.

Pandas

For tabular data, the data structure is called a "DataFrame" (aka. Table),

In [35]:
```
import pandas as pd
```

In [36]:
```
film_db = {
    'ratings': [7, 8, 9],
    'sweets': [12, 15, 29]
}
```

A generic variable name `df` is often used,

In [37]:
```
df = pd.DataFrame(film_db)
```

In [38]:
```
df
```

Out[38]:

	ratings	sweets
0	7	12
1	8	15
2	9	29

Q. Define your own dataframe called `health`, a healthcare dataset with `HeartRate` and `BloodPressure` columns.

NumPy

Pandas uses numpy. Numpy provides a fast "list" data structure, called an array.

In [39]:
```
y_like = [1, 1, 0, 1, 0, 0]
```

In [40]:
```
import numpy as np
```

In [41]:
```
array = np.array(y_like)
```

In [42]:
```
array # much faster than python's
```

Out[42]:
```
array([1, 1, 0, 1, 0, 0])
```

In [43]:
```
array.mean() # much faster than python's
```

Out[43]:
```
0.5
```

Seaborn

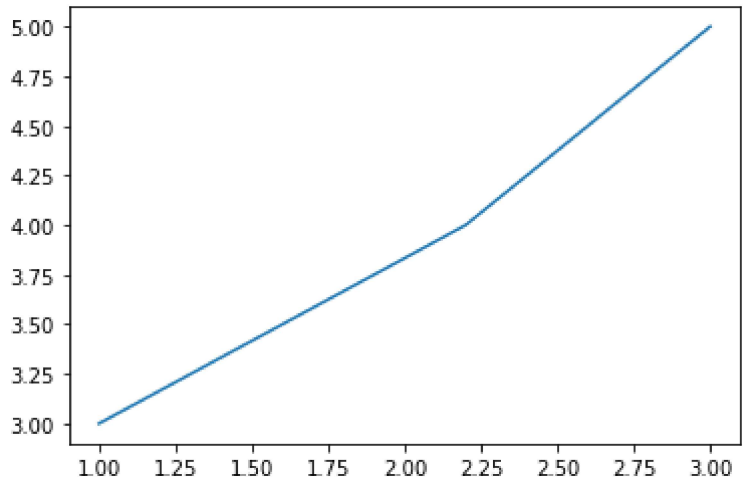
Seaborn is a simplified charting/plotting library,

In [44]:
```
# draw the plots in the notebook
%matplotlib inline

import seaborn as sns
```

In [45]:
```
sns.lineplot(x=[1, 2.2, 3], y=[3, 4, 5])
```

Out[45]:
```
<AxesSubplot:>
```



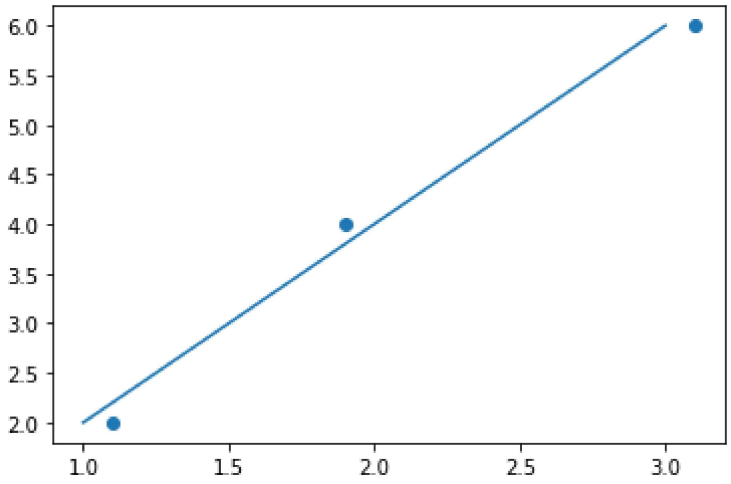
Q. with your health data, use `sns.scatterplot(x=.., y..)` to draw. Where `x=health['HeartRate']` and `y=health['BloodPressure']`.

matplotlib: the underlying library of seaborn

In [46]:
```
%matplotlib inline
import matplotlib.pyplot as plt
```

In [47]:
```
plt.plot([1, 2, 3], [2, 4, 6])
plt.scatter([1.1, 1.9, 3.1], [2, 4, 6])
```

Out[47]:
```
<matplotlib.collections.PathCollection at 0x2140cbad3a0>
```



Sci-Kit Learn

`sklean` is the mainstream simple machine learning library for python. As it is a collection of library, you almost always use `from sklearn import AMoreSpecificLibrary`.

In [48]:
```
from sklearn.linear_model import LinearRegression
```

In [49]:
```
df
```

Out[49]:

	ratings	sweets
0	7	12
1	8	15
2	9	29

All the Machine "Learning" happens in this line,

In [50]:
```
X = df[['sweets']]
y = df['ratings']

lm = LinearRegression().fit(X,y)
```

Q. Run `lm.score(X, y)` -- what does this tell you?

In [51]:
```
print("The slope is: ", lm.coef_[0].round(2))
print("The intercept is:", lm.intercept_.round(2))
```

```
The slope is:  0.1
```

```
The intercept is: 6.07
```

In [52]:

```
yhat_ratings = lm.predict([
    [10],
    [20]
])
```

```
C:\Users\Thomas Holmes\.conda\envs\ppds\lib\site-packages\sklearn\base.py:450: UserW
arning: X does not have valid feature names, but LinearRegression was fitted with fe
ature names
  warnings.warn(
```

In [53]:

```
print("Some ratings predictions, ", yhat_ratings)
```

```
Some ratings predictions,  [7.10526316 8.13765182]
```

Aside: polynomial regression, see https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.PolynomialFeatures.html