master DSLS

Programming 2

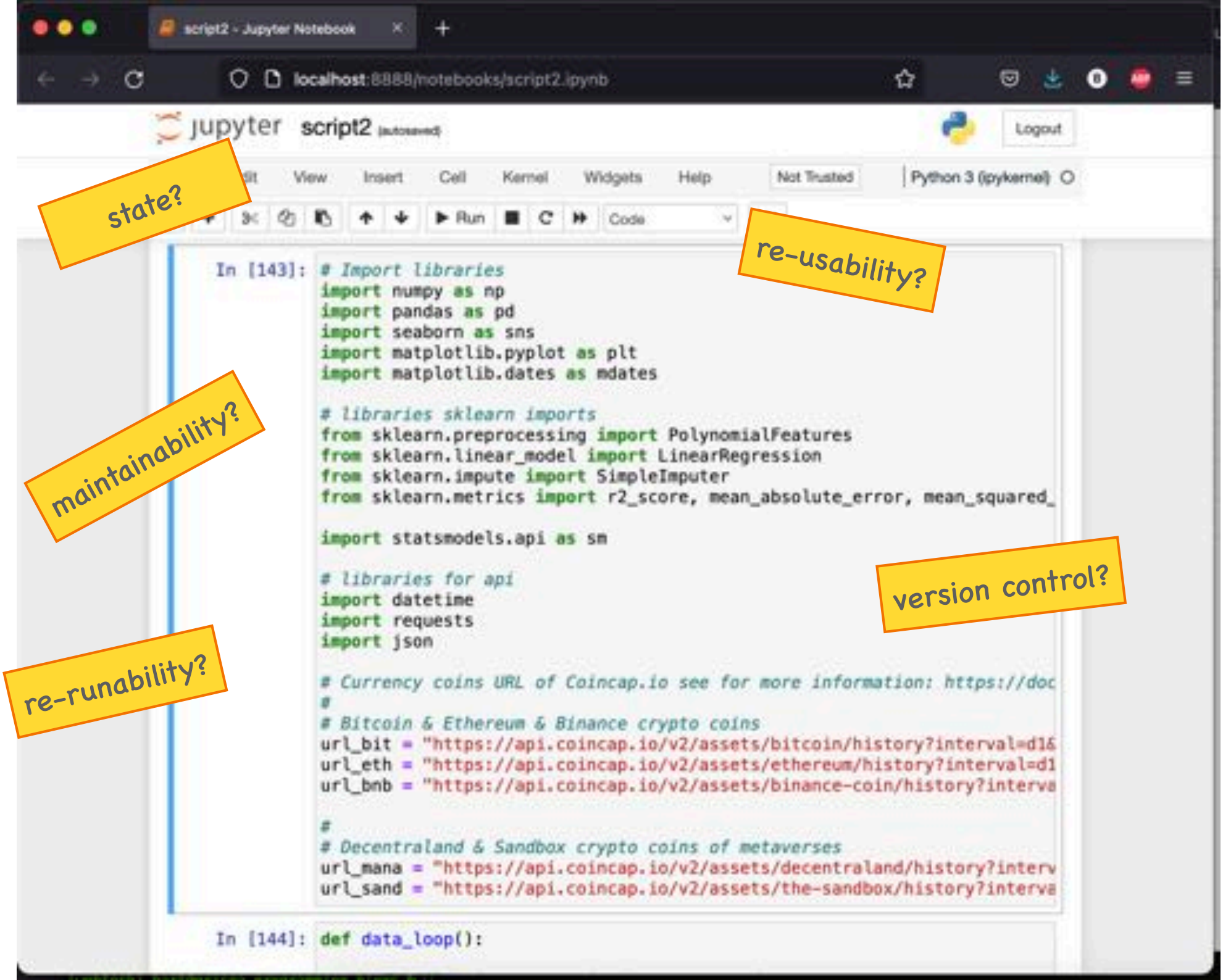
3. Multiple classes and interaction

Bart Barnard

Today's planning

12:30 - 13:00: recap and introduction
13:00 - 13:45: work on exercise 1
13:45 - 14:00: introduction exercise 2
14:00 - 14:30: break
14:30 - 15:00: work on exercise 2
15:00 - 15:30: recap / introduction exercise 3
15:30 - 16:15: work on exercise 3
16:15 - 16:30: generators

1. recap


script2 - Jupyter Notebook
localhost:8888/notebooks/script2.ipynb
jupyter script2 (autosaved)
Logout
View
Insert
Cell
Kernel
Widgets
Help
Not Trusted
Python 3 (ipykernel)
Run
Code
state?
re-usability?
maintainability?
version control?
re-runability?
In [143]:
# Import libraries
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
# libraries sklearn imports
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.impute import SimpleImputer
from sklearn.metrics import r2_score, mean_absolute_error, mean_squared_
import statsmodels.api as sm
# libraries for api
import datetime
import requests
import json
# Currency coins URL of Coincap.io see for more information: https://doc
#
# Bitcoin & Ethereum & Binance crypto coins
url_bit = "https://api.coincap.io/v2/assets/bitcoin/history?interval=d1&
url_eth = "https://api.coincap.io/v2/assets/ethereum/history?interval=d1
url_bnb = "https://api.coincap.io/v2/assets/binance-coin/history?interva
#
# Decentraland & Sandbox crypto coins of metaverses
url_mana = "https://api.coincap.io/v2/assets/decentraland/history?interv
url_sand = "https://api.coincap.io/v2/assets/the-sandbox/history?interva
In [144]:
def data_loop():


One cell with 450 LOC
16 methods, so on average 28 LOC/ method
longest line extends for 365 characters

2. multiple files and modules

file1.py

```
class Foo:
    def method1(self):
        # difficult things

    def method2(self):
        # other difficult things

class Bar:
    def method1(self):
        # bar's things

    def method2(self):
        # other bar's things
```

file2.py

```
from file1 import Foo, Bar

b = Foo()
b.method2()
```

file3.py

```
from file1 import Foo, Bar

b = Foo()
b.method2()
```

file1.py

```
class Foo:
    def method1(self):
        print ('called in method 1')

    def method2(self):
        print ('called in method 2')

class Bar:
    def method1(self):
        # bar's things

    def method2(self):
        # other bar's things

f = Foo()
f.method1()
```

file2.py

```
from file1 import Foo, Bar

b = Foo()
b.method2()
```

```
called in method 1
called in method 2
```

file1.py

```
class Foo:
    def method1(self):
        print ('called in method 1')

    def method2(self):
        print ('called in method 2')

class Bar:
    def method1(self):
        # bar's things

    def method2(self):
        # other bar's things

if __name__=='__main__':
    f = Foo()
    f.method1()
```

called in method 1

file2.py

```
from file1 import Foo, Bar

b = Foo()
b.method2()
```

called in method 2

Separation of concerns is a design principle for separating a computer program into distinct sections.

Each section addresses a separate concern, a set of information that affects the code of a computer program.

A concern can be as general as "the details of the hardware for an application", or as specific as "the name of which class to instantiate".

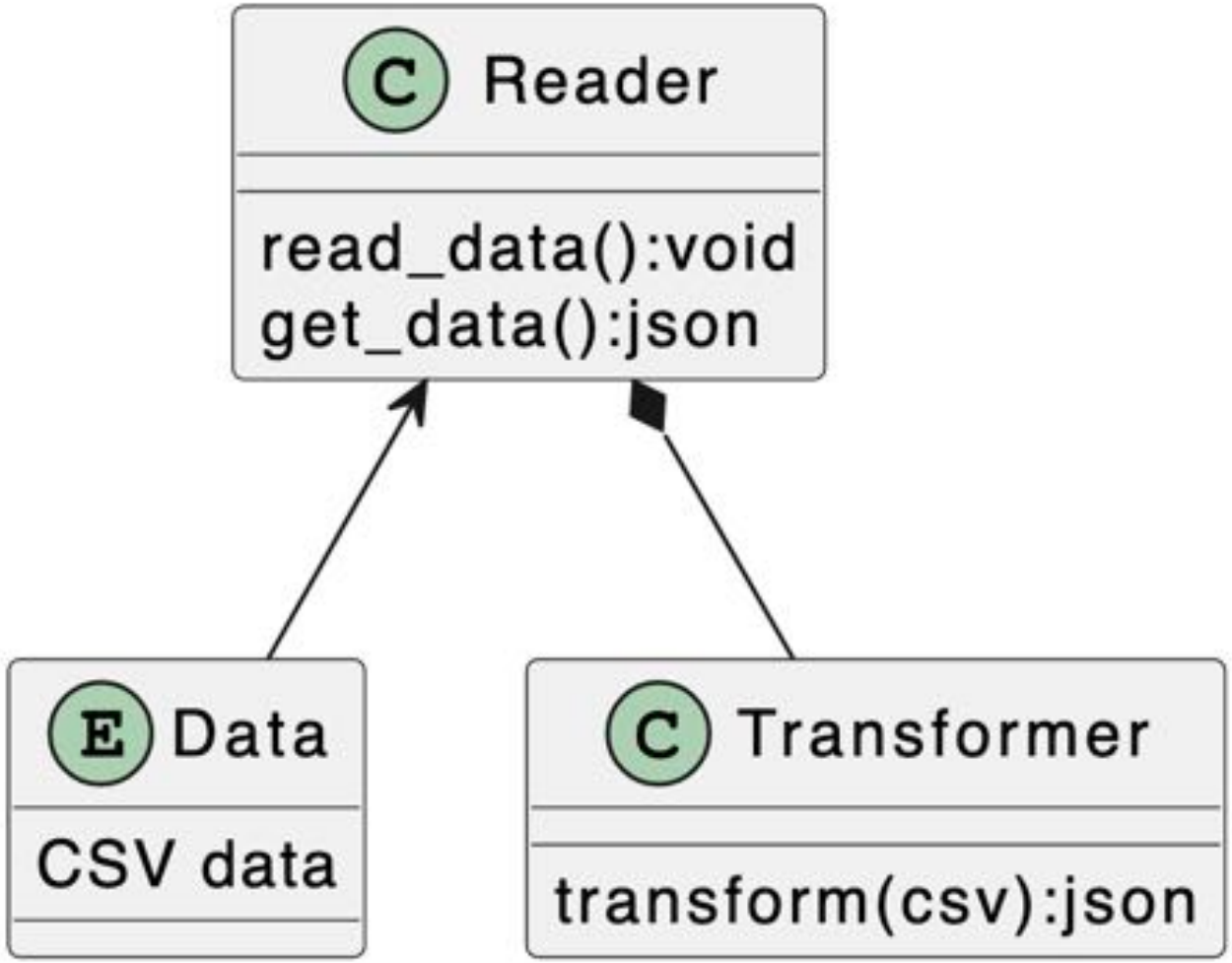
A program that embodies 'separation of concerns' well is called a modular program.

Modularity, and hence separation of concerns, is achieved by encapsulating information inside a section of code that has a well-defined interface.

Encapsulation is a means of information hiding.

https://en.wikipedia.org/wiki/Separation_of_concerns

Exercise 1 (45 mins)



When you're done, commit your code to github.

You are allowed to look at exercise 2 (and 3), but **not** allowed to work on them already.

file1.py

```
class Foo:
    def method1(self):
        print ('called in method 1')

    def method2(self):
        print ('called in method 2')

class Bar:
    def method1(self):
        # bar's things

    def method2(self):
        # other bar's things

if __name__=='__main__':
    f = Fo
    f.met
```

In the meantime keeping 'file1' more or less the same (open-close principle)

file2.py

```
from file1 import Foo, Bar

b = Foo()
b.metho
```

We want to be able to make use of 'file1' in lots of other files...

```
from file

b = Bar()
b.method2()
```

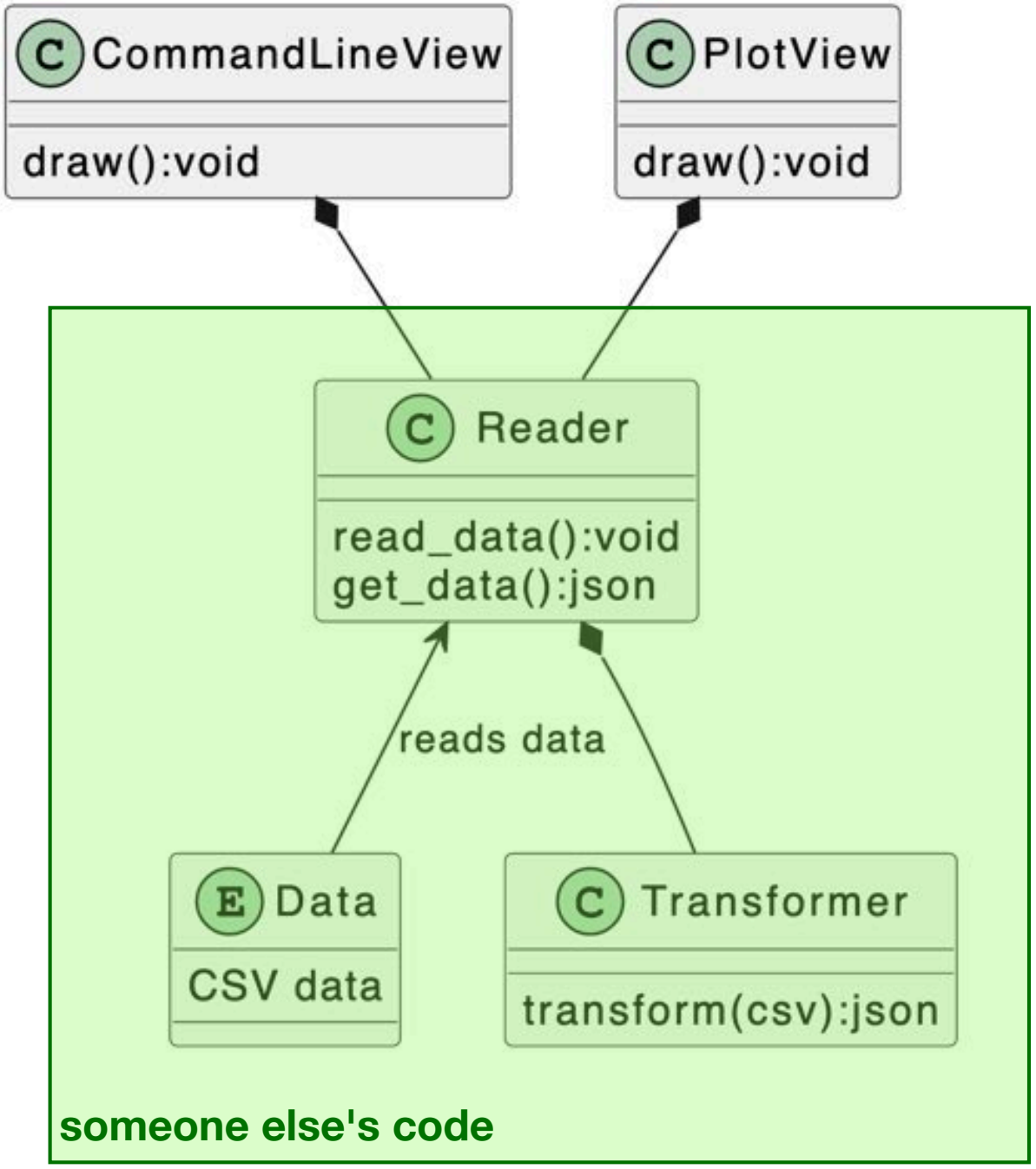
file4.py

```
from file1 import Foo, Bar

b = Bar()
f = Foo()
```

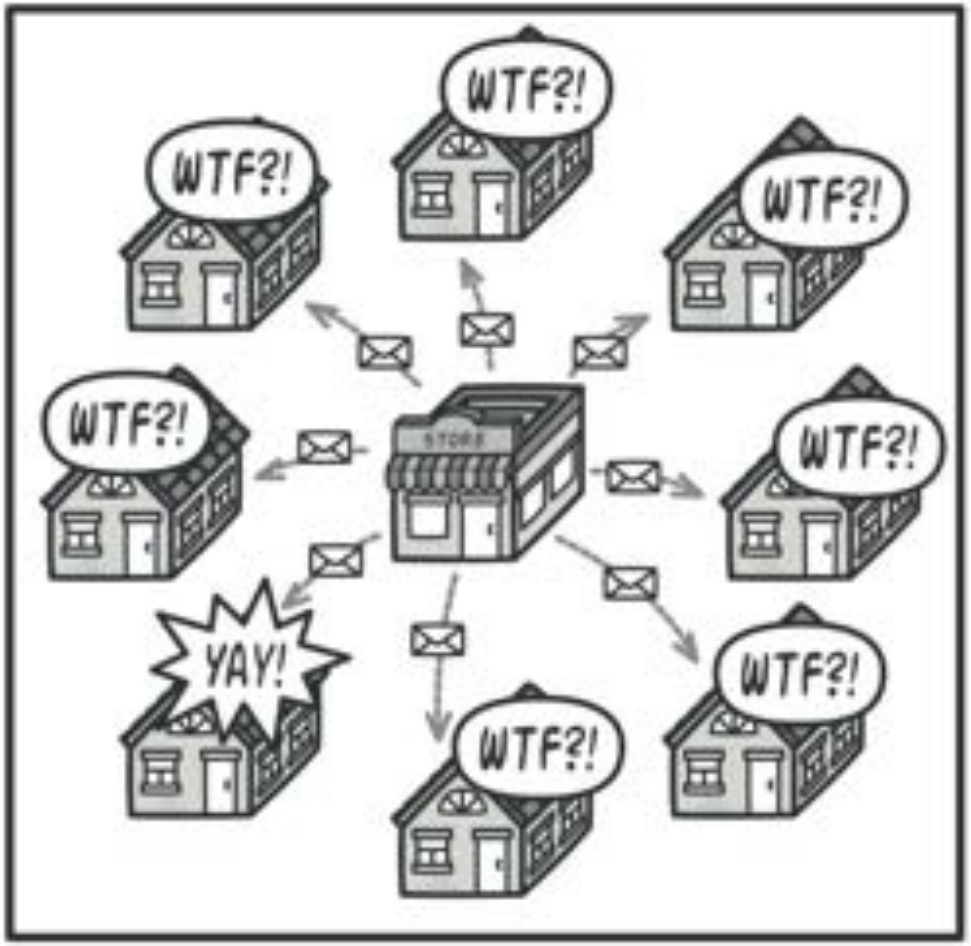
file5.py

```
from file1 import Foo, Bar

b = Foo()
b.method1()
```

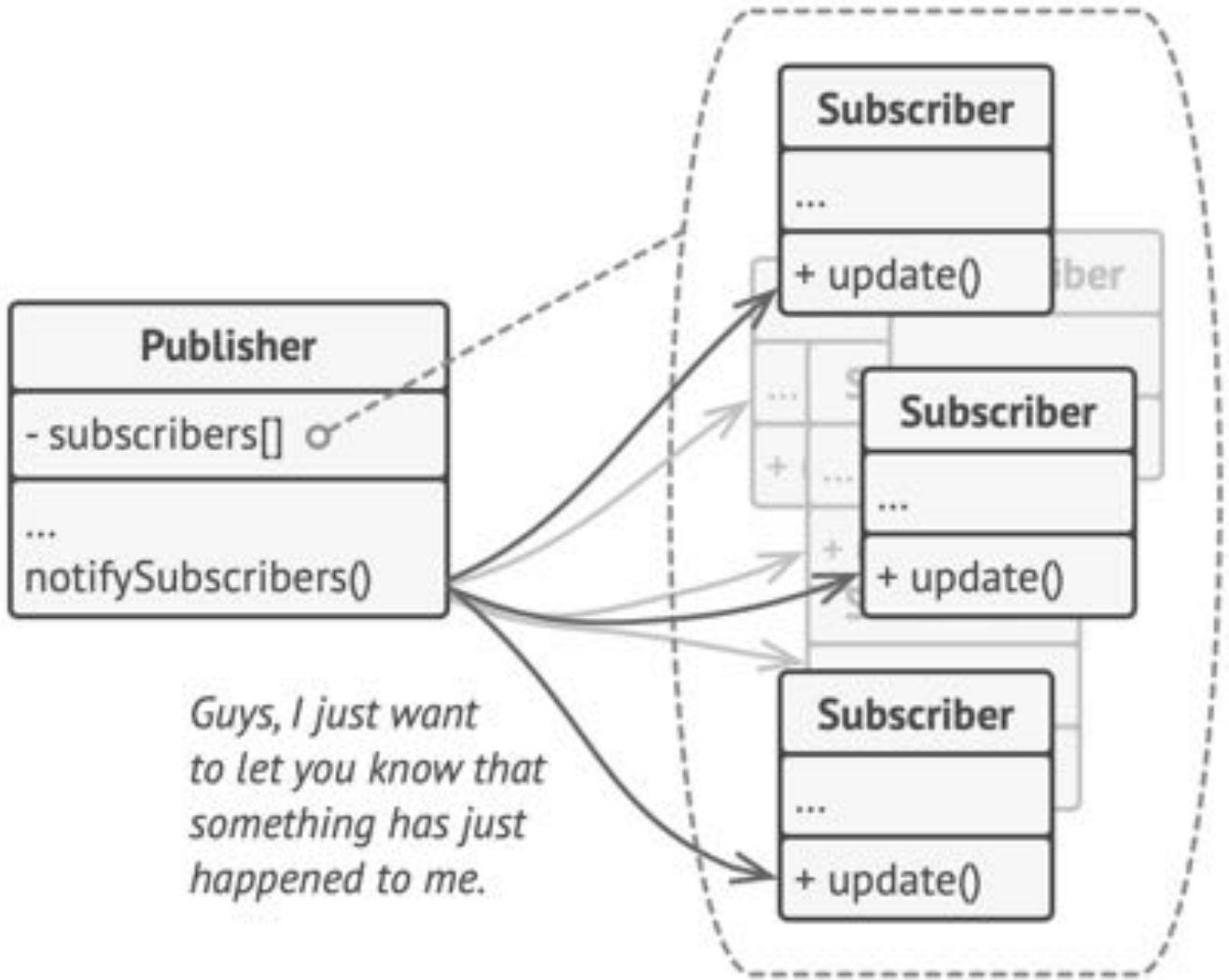
Exercise 2 (30 mins)



3. Observer pattern



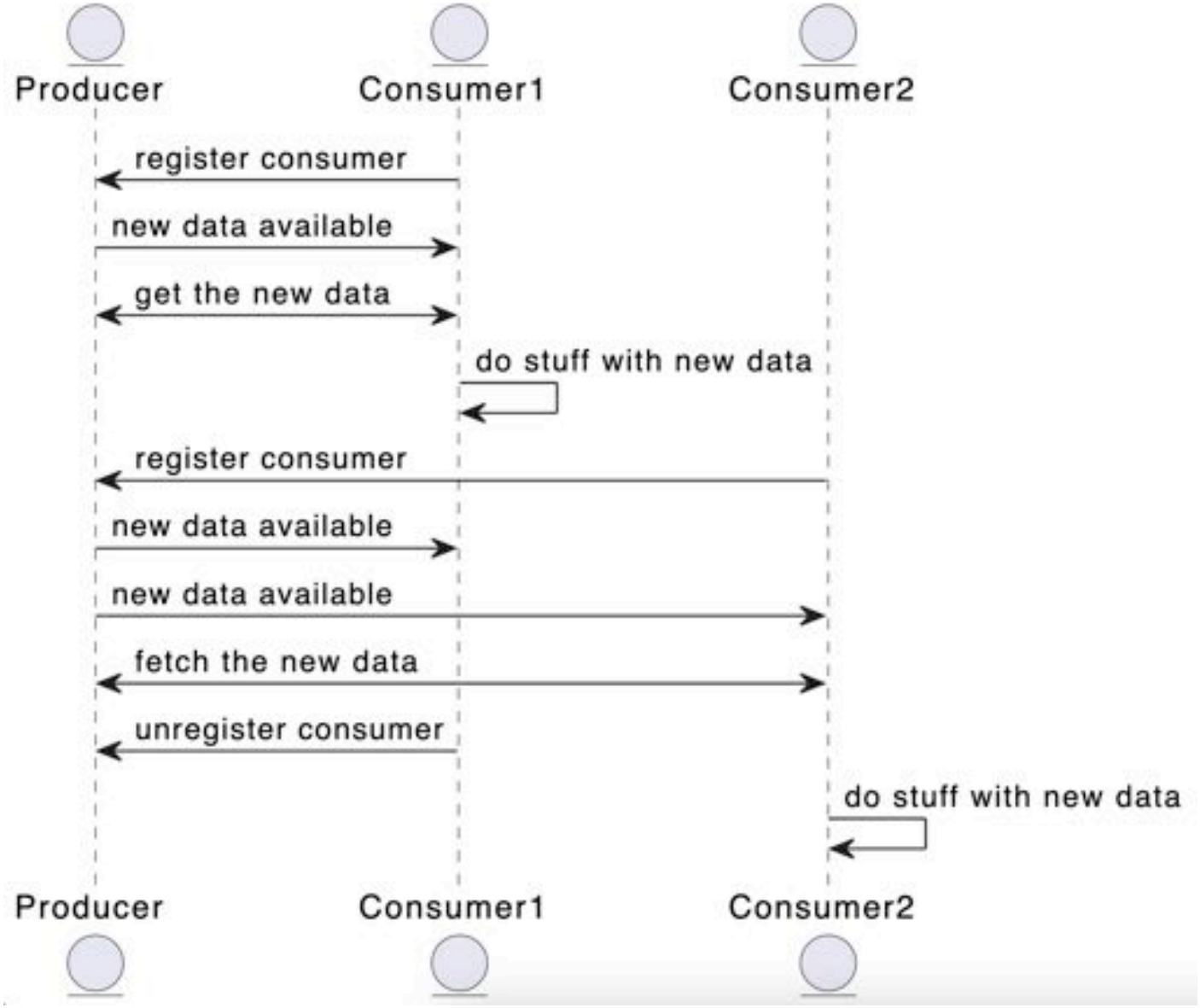
https://refactoring.guru/design-patterns/observer



Publisher notifies subscribers by calling the specific notification method on their objects.

https://refactoring.guru/design-patterns/observer

Exercise 3 (30 mins)



4. Generators ~~and async programming~~

```
def method():
  # do stuff
  return value
```

```
v = method()
```

```
def method():
  # do stuff
  yield value
  # do other stuff
  yield another_value
```

```
v = method()
v.__next__()
v.__next__()
```

```
v = method()
for x in v:
    # do stuff with returned value
```