Session 14

Pepe García

September 21, 2021

In this **Async session** you'll go over a series of exercises to learn how to model data depending on the problem at hand.

Being able to select the correct datatype for each use case is one of the most important skills we'll need to program efficiently.

Exercises

In this series of exercises we´ll see how different problems require different data structures.

There are solutions for this problems in the Solutions section, but try to work on these examples without looking at them beforehand.

DNS

In this first example we'll try to model what a DNS server does.

For this exercise we need to first create a *database* that contains the mappings from domain names to ip addresses. This *database* we'll save it in a variable called `database` and should contain the following mappings:

Domain name	**IP address**
amazon.com	10.32.23.111
google.com	10.255.23.111
wordpress.com	253.32.23.31

Once you have your `database` created, define a function called `resolve_dns` that allows you to translate from a domain name to a IP address, and if the domain name is unknown, returns `None`.

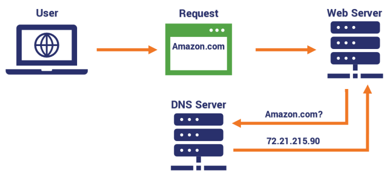


Figure 1:

THE DOMAIN NAME SYSTEM is the subsystem of the Internet in charge of translating from domain names to IP addresses. Each computer connected to the Internet has an IP address associated so that other computers can refer to it. These addresses look like **102.43.250.21**, making it fairly hard to remember them all.

Luckily, **DNS** allows us to map domain names, such as **google.com** to IP addresses like **102.43.250.21**

Matrices

Matrices mathematical objects composed by smaller elements and distributed in rows and columns.

$$\begin{array}{c|cccc|} & 1 & 2 & \dots & n \\ 1 & a_{11} & a_{12} & \dots & a_{1n} \\ 2 & a_{21} & a_{22} & \dots & a_{2n} \\ 3 & a_{31} & a_{32} & \dots & a_{3n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ m & a_{m1} & a_{m2} & \dots & a_{mn} \end{array}$$

Figure 2: This image shows a $m * n$ matrix.

In order to add matrices together we need to sum elements position wise. Given

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5 \end{pmatrix}$$

and

$$B = \begin{pmatrix} 4 & 5 & 6 \\ 5 & 6 & 7 \\ 6 & 7 & 8 \end{pmatrix}$$

Then:

$$A + B = \begin{pmatrix} 5 & 7 & 9 \\ 7 & 9 & 11 \\ 9 & 11 & 13 \end{pmatrix}$$

We calculate each position in the sum matrix as follows:

$$(A + B)_{ij} = A_{ij} + B_{ij}$$

Now, try to implement matrix addition using Python. First we need to model matrices in a way that allows us to use them.

Let's create two variables, `a` and `b` that contain the same matrices we've declared above.

Afterwards, create a function `matrix_addition(a, b)` that performs matrix addition as we know how to do it. This function should return the solution matrix.

Solutions

Solution for DNS

```
database = {
  "amazon.com": "10.32.23.111",
  "google.com": "10.255.23.111",
  "wordpress.com": "253.32.23.31"
}

def resolve_dns(domain):
    # if ... in  checks if an element (domain in this case) exists in
    # the keys of a dictionary
    if domain in database:
        # If it exists, we return it
        return database[domain]
    else:
        # Otherwise, we return None, the empty value
        return None
```

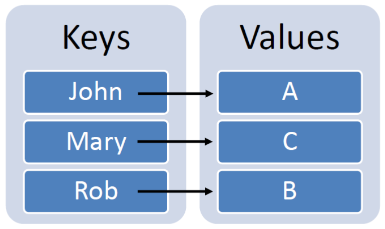
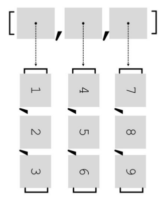


Figure 3: In this exercise it makes sense to use a **dictionary** because we're mapping between two sets. Domain names and IP addresses.

Each one of the entries in our *database* becomes an entry in our **dictionary**, with a **key** for the domain name and a **value** for the IP address

Solution for Matrices

```
A = [
  [1, 2, 3],
  [2, 3, 4],
  [3, 4, 5]
]

B = [
  [4, 5, 6],
  [5, 6, 7],
  [6, 7, 8]
]

def matrix_addition(a, b):
    # We start with an empty list for our solution matrix
    solution = []

    # i is the index for our rows
    i = 0

    # here we’re iterating over all rows
    while i < len(a):
        # We start by creating an empty row
        solution.append([])

        # j is the index for our columns
        j = 0

        # here we’re iterating over all columns in a row
        while j < len(a[i]):
            # At this point we use the method we know from matrix addition
            # to sum this position on a and b matrices.
            solution[i].append(a[i][j] + b[i][j])
            j = j + 1
        i = i + 1

    return solution
```



Python List of Lists

```
>>> lst_1 = [1, 2, 3]
>>> lst_2 = [4, 5, 6]
>>> lst_3 = [7, 8, 9]
>>> lst = [lst_1, lst_2, lst_3]
>>> lst
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Figure 4: For this problem, it makes sense to use **lists of lists** to represent matrices. Lists can be contained within other lists in Python.

Since the size of a matrix is unbounded, they can start at zero rows and columns up to an infinite number of rows and columns, it makes sense to use lists, which allow us to make them grow as much as we need.