L9d: Advanced Data Structures: Dictionaries

Motivation

- How could we store the information needed to manage students enrolled in this course?

NIU	Cognom	Nom	Nota
123456	Sagarra	Jesus	5.6
123465	Lacruz	Anna	7.8
234567	Mento	Lola	4.5
478933	Bronca	Armand	9.7

- One option is to create a list for each column:

```
NIU =[123456, 123465, 234567, 478933]
Cognom= [‘Sagarra’, ‘Lacruz’, ‘Mento’,’Bronca’]
Nom = [‘Jesus’, ‘Anna’, ‘Lola’, ‘Armand’]
Nota = [5.6, 7.8, 4.5, 9.7]
```

Motivation

- To write a function that tells us a student's grade based on their surname, we should do:

```
def Obtenir_Nota(Cognom,Llista_Cognom,Llista_Nota):
    i = Llista_Cognom.index(Cognom)
    nota = Llista_Nota[i]
    return(nota)
```

- But doing it in this way, implies :
  - Introduce a certain clutter if there is a lot of different information to manage.
  - Maintain many lists and pass them as arguments.
  - Index using integers.
  - Remember to make changes to multiple lists..

Dictionaries

- It would be more pleasant to directly indicate the item of interest (not always `int`).
- It is better to use a single data structure, not separate lists.

A List

0	Element0
1	Element1
2	Element2
3	Element3
…	…

Index Element

A Dictionary

Key1	Value1
Key2	Value2
Key3	Value3
Key4	Value4
…	…

Personalized index, using a label Element

Dictionaries: Definition

- We store pairs of data:
  - Key
  - Value

'Sagarra'	5.6
'Lacruz'	7.8
'Mento'	4.5
'Bronca'	9.7
…	…

- The syntax for expressing dictionary-type literals is:
  - The key, followed by colon (**:**) to separate it from the value
  - The value
  - Each element is separated using comas (**,**) inside curly brackets **{}**

```
Nota = {'Sagarra':5.6, 'Lacruz':7.8, 'Mento':4.5, 'Bronca':9.7}
```

Key1 Element1 Key2 Element2 Key3 Element3 Key4 Element4

Indexing and Access

- We access to an element of the dictionary using the operator **[ ]** with the **key** inside:

```
In [57]: Nota = {'Sagarra': 5.6, 'Lacruz': 7.8, 'Mento': 4.5, 'Bronca': 9.7}

In [58]: Nota['Lacruz']
Out[58]: 7.8
```

- If the key is not in the dictionary, it returns an error:

```
In [59]: Nota['Salero']
Traceback (most recent call last):

  File "<ipython-input-59-a45173151646>", line 1, in <module>
    Nota['Salero']

KeyError: 'Salero'
```

Dictionary: Definition

- A dictionary is a **Non-ordered sequence** of values, where each value is identified using a **key** (not an index).

- The elements in the dictionary can be **heterogenous**.

- It is **Mutable**, we can change its values.

- Variable size.

- It can be nested (dictionaries inside a dictionary).

```
In [60]: llistat = {123456: {'cognom': 'Sagarra', 'nom':'Jesus', 'Nota': 5.6},
    ...:            123465: {'cognom': 'Lacruz', 'nom':'Anna', 'Nota': 7.8},
    ...:            234567: {'cognom': 'Mento', 'nom':'Lola', 'Nota': 4.5},
    ...:            478933: {'cognom': 'Bronca', 'nom':'Armand', 'Nota': 9.7} }
```

Access to a nested dictionary

- We first access to the dictionary using the key:

```
In [63]: llistat[123456]
Out[63]: {'cognom': 'Sagarra', 'nom': 'Jesus', 'Nota': 5.6}
```

- Then we concatenate the brackets **[ ]** to access to each element in the nested dictionary (in this case, we access to the key of the nested dictionary):

```
In [64]: llistat[123456]['Nota']
Out[64]: 5.6
```

Changing elements in a dictionary

- Dictionaries are **Mutable**
- We access to the element with the key and bracket **[ ]** and assign the new value with the operator **=** :

```
In [64]: llistat[123456]['Nota']
Out[64]: 5.6
```

```
In [65]: llistat[123456]['Nota'] = 6.5

In [66]: llistat
Out[66]:
{123456: {'cognom': 'Sagarra', 'nom': 'Jesus', 'Nota': 6.5},
 123465: {'cognom': 'Lacruz', 'nom': 'Anna', 'Nota': 7.8},
 234567: {'cognom': 'Mento', 'nom': 'Lola', 'Nota': 4.5},
 478933: {'cognom': 'Bronca', 'nom': 'Armand', 'Nota': 9.7}}
```

Keys and values

Keys	Values
It must be **unique**	Duplicates are allowed
It must be of a **immutable** type (`int, float, string, tuple,bool`)	It can be of any type (both mutables and immutables)
Be careful with the keys of type `float` (due to the differences in precision)	The values of the dictionary can be lists, or even other dictionaries
Not ordered	Not ordered

Lists vs Dictionaries

Lists	Dictionaries
Ordered sequence of elements	It matches keys with values
Indexes are ordered	The order is not guaranteed
It searches an element from an index (integer value)	It searches an element from another element (not only integers)
The index is an integer	The key can be any immutable element

Operations with dictionaries: Add and Delete

- To add an item into the dictionary, we will assign a new key:

```
In [91]: Nota
Out[91]: {'Sagarra': 5.6, 'Lacruz': 7.8, 'Mento': 4.5, 'Bronca': 9.7}

In [92]: Nota['Larraz'] = 5.1

In [93]: Nota
Out[93]: {'Sagarra': 5.6, 'Lacruz': 7.8, 'Mento': 4.5, 'Bronca': 9.7, 'Larraz': 5.1}
```

- To delete an item from a dictionary we will use the function `del`:

```
In [94]: del(Nota['Larraz'])

In [95]: Nota
Out[95]: {'Sagarra': 5.6, 'Lacruz': 7.8, 'Mento': 4.5, 'Bronca': 9.7}
```

But if the element does not exist, it will raise an error

Other operations

- To empty a dictionary we use the method `clear`:

```
In [95]: Nota
Out[95]: {'Sagarra': 5.6, 'Lacruz': 7.8, 'Mento': 4.5, 'Bronca': 9.7}

In [96]: Nota.clear()

In [97]: Nota
Out[97]: {}
```

- Dictionaries are mutable. To make a copy, we use `copy`:

```
In [98]: Nota = {'Sagarra': 5.6, 'Lacruz': 7.8, 'Mento': 4.5, 'Bronca': 9.7}

In [99]: CopiaNota = Nota.copy()

In [100]: Nota['Larraz'] = 5.1

In [101]: CopiaNota
Out[101]: {'Sagarra': 5.6, 'Lacruz': 7.8, 'Mento': 4.5, 'Bronca': 9.7}

In [102]: Nota
Out[102]: {'Sagarra': 5.6, 'Lacruz': 7.8, 'Mento': 4.5, 'Bronca': 9.7, 'Larraz': 5.1}
```

Operations: Length, Membership

- `len(d)`, returns the length of the dictionary

- The structure is not ordered, so we can **not** use the operator `[n:m]`.

- The operation `in` checks if the key appears in the dictionary.

```
In [75]: Nota = {'Sagarra': 5.6, 'Lacruz': 7.8, 'Mento': 4.5, 'Bronca': 9.7}

In [76]: 'Mento' in Nota
Out[76]: True

In [77]: 'Rodolfo' not in Nota
Out[77]: True
```

Iteration

- In a dictionary we can work with the keys, the values or both:

```
Nota = {'Sagarra':5.6, 'Lacruz':7.8, 'Mento':4.5, 'Bronca':9.7}
```

```
increment = 1
for i in Nota:
    Nota[i] = Nota[i]+ increment
```

```
Nota = {'Sagarra’:6.6, 'Lacruz’:8.8, 'Mento’:5.5, 'Bronca’:10.7}
```

Operations: Keys, Values, Items

- To obtain an iterable tuple with all the **keys** we can use the method `keys()`. It is not guaranteed to obtain them ordered:

```
In [85]: Nota
Out[85]: {'Sagarra': 5.6, 'Lacruz': 7.8, 'Mento': 4.5, 'Bronca': 9.7}

In [86]: Nota.keys()
Out[86]: dict_keys(['Sagarra', 'Lacruz', 'Mento', 'Bronca'])
```

- To obtain an iterable tuple with all the **values** we can use the method `values()`. It is not guaranteed to obtain them ordered:

```
In [87]: Nota.values()
Out[87]: dict_values([5.6, 7.8, 4.5, 9.7])
```

- To obtain an iterable tuple with all the pairs key-value we can use the method `items()`:

```
In [90]: Nota.items()
Out[90]: dict_items([('Sagarra', 5.6), ('Lacruz', 7.8), ('Mento', 4.5), ('Bronca', 9.7)])
```

Iteration

```
Nota = {'Sagarra':5.6, 'Lacruz':7.8, 'Mento':4.5, 'Bronca':9.7}
```

- keys()

```
Increment = 1
for i in Nota.keys():
    Nota[i] = Nota[i]+ Increment
```

```
Nota = {'Sagarra’:6.6,
'Lacruz’:8.8, 'Mento’:5.5,
'Bronca’:10.7}
```

- values()

```
Ap = 0
Sus = 0
for i in Nota.values():
    if i >= 5:
        Ap += 1
    else
        Sus += 1
print(“Passed: ", Ap," Failed:", Sus)
```

```
Passed: 3 Failed: 1
```

- items()

```
Increment = 1
for k,v in Nota.items():
    Nota[k] = v + Increment
```

```
Nota = {'Sagarra’:6.6,
'Lacruz’:8.8, 'Mento’:5.5,
'Bronca’:10.7}
```

Exercise: Frequence of a word

- We want to know which is the most repeated word in the song *Bohemian Rhapsody (Queen)*
- Reuse the function *Lyrics2list*.
- Create two functions:
  - `Lyrics2frequencies(list)`. This function will be used to know the frequency of occurrence of each word. It will receive a list of strings (words) and return a dictionary where the key will be the word and the value will be the number of times that word appears in the text.
  - `Most_common_words(frequencies)`. This function will search for the most repeated words. It will receive a dictionary (from the previous function) and will calculate the maximum repetition value, returning two parameters:
    - The list of the most repeated words.
    - The maximum value (int) in the list

Exercise: Frequence of a word

- We want to know which is the most repeated word in the song *Bohemian Rhapsody (Queen)*

```
def Lyrics2frequencies(llista):
    myDict = {}
    for word in llista:
        if word in myDict:
            myDict[word] += 1
        else:
            myDict[word] = 1
    return myDict

def Most_common_words (freqs):
    best = max(freqs.values())
    words = []
    for k in freqs:
        if freqs[k] == best:
            words.append(k)
    return (words, best)
```

Dictionary comprehension

- An elegant/compact way to create a dictionary from another one

```
dict = {key:value for (key,value) in dictonary.items()}
```

- For this, we can work with the keys, the values or both:

```
Nota = {'Sagarra':5.6, 'Lacruz':7.8, 'Mento':4.5, 'Bronca':9.7}
```

```
increment = 1
for i in Nota:
    Nota[i] = Nota[i]+ increment
```

Equivalent code

```
Nota = {i:v+1 for (i,v) in Nota.items() }
```

```
Nota = {'Sagarra’:6.6, 'Lacruz’:8.8, 'Mento’:5.5, 'Bronca’:10.7}
```

Dictionary comprehension

- An elegant/compact way to create a dictionary from another one

```
dict = {key:value for (key,value) in dictonary.items()}
```

- Example A: double each **value** in the dictionary:

```
dict1 = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}

double_dict1 = {k:v*2 for (k,v) in dict1.items()}
print(double_dict1)

OUT -> {'e': 10, 'a': 2, 'c': 6, 'b': 4, 'd': 8}
```

- Example B: double the **key** values in the dictionary

```
dict1 = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}

dict1_keys = {k*2:v for (k,v) in dict1.items()}
print(dict1_keys)

OUT -> {'aa': 1, 'bb': 2, 'cc’: 3, 'dd': 4, 'ee': 5}
```

Dictionary comprehension

- An elegant/compact way to create a dictionary from another one

```
dict = {key:value for (key,value) in dictonary.items()}
```

- It can substitute `lambda` functions

```
fahrenheit = {'t1':-30, 't2':-20, 't3':-10, 't4':0}

celsius = list(map(lambda x: (float(5)/9)*(x-32), fahrenheit.values()))

#Create the `celsius` dictionary
celsius_dict = dict(zip(fahrenheit.keys(), celsius))

print(celsius_dict)
OUT → {'t2': -28.88, 't3': -23.33, 't1': -34.44, 't4': -17.77}
```

Equivalent code

```
celsius2 = {k:(float(5)/9)*(v-32) for (k,v) in fahrenheit.items()}

print(celsius2)
OUT → {'t2': -28.88, 't3': -23.33, 't1': -34.44, 't4': -17.77}
```

Dictionary comprehension

- An elegant/compact way to create a dictionary from another one

```
dict = {key:value for (key,value) in dictonary.items()}
```

- It can work with conditionals

```
dict1 = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}

#Select only items greater than 2
dict1_cond = {k:v for (k,v) in dict1.items() if v>2}

print(dict1_cond)
OUT → {'c': 3, 'd': 4, 'e': 5}
```

- It can have several `if` conditions

```
#Select items greater than 2 and multiples of 2
dict1_doubleCond = {k:v for (k,v) in dict1.items() if v>2 if v%2==0}

print(dict1_doubleCond)
OUT → {'d': 4}
```