

Lesson 4 – Data structures:

Lists

Lists: Definition

A list is an ordered sequence of values, where each value is identified by an index.

The elements in the list can be heterogeneous.

To define the elements in a list, we write the elements in square brackets separated by a comma:

```
In [1]: my_list=["Bohemian Rhapsody", 1975, "Queen"]
```

```
In [2]: type(my_list)
```

```
Out[2]: list
```

Empty list:

```
In [3]: my_list=[]
```

```
In [4]: type(my_list)
```

```
Out[4]: list
```

Lists: Definition

It is possible to create a **list of lists**:

```
In [5]: my_list=["Bohemian Rhapsody", 1975, "Queen",["art rock","opera","hard rock"]]
```

```
In [6]: my_list
```

```
Out[6]: ['Bohemian Rhapsody', 1975, 'Queen', ['art rock', 'opera', 'hard rock']]
```

First item: 'Bohemian Rhapsody'

Second item: 1975

Third item: 'Queen'

Forth item: ['art rock', 'opera', 'hard rock']

Indexing and access

We access an item in the list using the **operator []** and an **index** corresponding to the relative position of the item in the list.

The first item has index 0.

```
In [6]: my_list
Out[6]: ['Bohemian Rhapsody', 1975, 'Queen', ['art rock', 'opera', 'hard rock']]

In [7]: len(my_list)
Out[7]: 4

In [8]: my_list[0]
Out[8]: 'Bohemian Rhapsody'

In [9]: my_list[1]+1
Out[9]: 1976

In [10]: my_list[3]
Out[10]: ['art rock', 'opera', 'hard rock']

In [11]: i=1

In [12]: my_list[i+1]
Out[12]: 'Queen'
```

Access to the last items in a list

It is possible to access the items from the end by using negative indexes.

In this case, the last item in a list can be accessed with index -1, second-to-last with index -2, and so on:

```
In [13]: my_list
Out[13]: ['Bohemian Rhapsody', 1975, 'Queen', ['art rock', 'opera', 'hard rock']]

In [14]: my_list[-1]
Out[14]: ['art rock', 'opera', 'hard rock']

In [15]: my_list[-4]
Out[15]: 'Bohemian Rhapsody'
```

Access to a nested list

First, we access the whole nested list using its index and the operator []:

```
In [16]: my_list
Out[16]: ['Bohemian Rhapsody', 1975, 'Queen', ['art rock', 'opera', 'hard rock']]

In [17]: my_list[3]
Out[17]: ['art rock', 'opera', 'hard rock']
```

Next, we use the operator [] again to access each element of the nested list:

```
In [18]: my_list[3][1]
Out[18]: 'opera'
```

Index outside the list

Trying to access an item beyond the length of the list, will cause an **IndexError** exception (error).

```
In [19]: my_list
```

```
Out[19]: ['Bohemian Rhapsody', 1975, 'Queen', ['art rock', 'opera', 'hard rock']]
```

```
In [20]: my_list[4]
```

```
Traceback (most recent call last):
```

```
File "<ipython-input-20-d8d3b5435ffd>", line 1, in <module>  
    my_list[4]
```

```
IndexError: list index out of range
```

```
In [21]:
```

```
In [21]: my_list[-5]
```

```
Traceback (most recent call last):
```

```
File "<ipython-input-21-80aece02dfaa>", line 1, in <module>  
    my_list[-5]
```

```
IndexError: list index out of range
```

Change the items in a list

Lists are **mutable** → Their elements can be modified.

Each position in a list acts as a separate variable.

We access the position with the operator [] and assign the new value with the operator = .

```
In [22]: my_list
```

```
Out[22]: ['Bohemian Rhapsody', 1975, 'Queen', ['art rock', 'opera', 'hard rock']]
```

```
In [23]: my_list[1]=1977
```

```
In [24]: my_list
```

```
Out[24]: ['Bohemian Rhapsody', 1977, 'Queen', ['art rock', 'opera', 'hard rock']]
```


Length, cut, membership

`len(l)` returns the length of the list:

```
In [25]: my_list=[1,2,3,4,5,6,7,8,9]
```

```
In [26]: len(my_list)
```

```
Out[26]: 9
```

The operator `[n:m]` returns the sub-list from the n-th element (included) to the m-th (not included):

```
In [27]: my_list[3:7]
```

```
Out[27]: [4, 5, 6, 7]
```

The operator `in` checks if an item is in the list. It returns a Boolean. It can be combined with operator `not`.

```
In [28]: 10 in my_list
```

```
Out[28]: False
```

```
In [30]: 10 not in my_list
```

```
Out[30]: True
```

```
In [29]: 5 in my_list
```

```
Out[29]: True
```

```
In [31]: 5 not in my_list
```

```
Out[31]: False
```

List operations: Adding items

To add an item at the end of the list we use `append`:

```
In [34]: my_list=[1,3,10,8,7,5,2,0,2]
```

```
In [35]: my_list.append(12)
```

```
In [36]: my_list
```

```
Out[36]: [1, 3, 10, 8, 7, 5, 2, 0, 2, 12]
```

To add an element x at a certain position i , we use `insert(i, x)`:

```
In [37]: my_list=[1,3,10,8,7,5,2,0,2]
```

```
In [38]: my_list.insert(3,99)
```

```
In [39]: my_list
```

```
Out[39]: [1, 3, 10, 99, 8, 7, 5, 2, 0, 2]
```



Item 99 at position 3

List operations: Adding items

We can concatenate two or more lists with the operator `+`, returning a new list:

```
In [40]: my_list=[1,2,3,4,5,6,7,8,9]
```

```
In [41]: new_list = my_list[3:7] + ['a','b','c']
```

my_list has not changed

```
In [42]: new_list
```

```
Out[42]: [4, 5, 6, 7, 'a', 'b', 'c']
```

To concatenate we also have `extend`:

```
In [43]: my_list=[1,2,3,4,5,6,7,8,9]
```

```
In [44]: my_list.extend([10,11,12])
```

my_list has changed

```
In [45]: my_list
```

```
Out[45]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

What happens if we use `append`?

```
In [46]: my_list=[1,2,3,4,5,6,7,8,9]
```

```
In [47]: my_list.append([10,11,12])
```

my_list is a nested list

```
In [48]: my_list
```

```
Out[48]: [1, 2, 3, 4, 5, 6, 7, 8, 9, [10, 11, 12]]
```

List Operations: Delete

To remove an item at the end of the list we have `pop()`. It returns the deleted item and changes the list:

```
In [49]: my_list=[1,2,3,4,5,6,7,8,9]
```

```
In [50]: x=my_list.pop()
```

```
In [51]: x
```

```
Out[51]: 9
```

```
In [52]: my_list
```

```
Out[52]: [1, 2, 3, 4, 5, 6, 7, 8]
```

To remove an item from a certain position in the list we use function `del`:

```
In [53]: my_list
```

```
Out[53]: [1, 2, 3, 4, 5, 6, 7, 8]
```

```
In [54]: del(my_list[2])
```

```
In [55]: my_list
```

```
Out[55]: [1, 2, 4, 5, 6, 7, 8]
```

List Operations: Delete

To remove an item with a certain value we have `remove`:

- it searches for the item and removes it
- if the item appears multiple times, only removes the first occurrence
- if the item is not in the list, it generates an error

```
In [56]: my_list = [2,1,3,6,3,7,0]
```

```
In [57]: my_list.remove(7)
```

```
In [58]: my_list
```

```
Out[58]: [2, 1, 3, 6, 3, 0]
```

```
In [59]: my_list.remove(3)
```

```
In [60]: my_list
```

```
Out[60]: [2, 1, 6, 3, 0]
```

```
In [61]: my_list.remove(9)
```

```
Traceback (most recent call last):
```

```
File "<ipython-input-61-f1a5247a45bf>", line 1, in <module>
    my_list.remove(9)
```

```
ValueError: list.remove(x): x not in list
```

Iteration on a list (`for`)

Given the following list of integers:

```
In [62]: my_list = [1,3,10,8,7,5,2,0,2]
```

```
In [63]: my_list
```

```
Out[63]: [1, 3, 10, 8, 7, 5, 2, 0, 2]
```

We want to make a program to sum all the values in the list:

```
summation = 0
for i in range(len(my_list)):
    summation += my_list[i]
print("The sum is: ", summation)
```

Notes:

- The elements in the list are indexed from 0 to `len(my_list) - 1`
- Function `range(n)` generates values from 0 to `n-1`

```
summation = 0
for element in my_list:
    summation += element
print("The sum is: ", summation)
```

Notes:

- We can **iterate directly on the items in the list**.
- In strings we can also iterate on their elements.


This code is more 'Pythonian'

Iterate on a list (for)

A list **CAN NOT** be modified while it is being iterated.

Example: Function that removes items that are in another list

```
def remove_duplicates(list_1, list_2):  
    for e in list_1:  
        if e in list_2:  
            list_1.remove(e)
```




```
In [68]: list_1 = [1,2,3,4]
```

```
In [69]: list_2 = [1,2,5,6]
```

```
In [70]: remove_duplicates(list_1, list_2)
```

```
In [71]: print(list_1)  
[2, 3, 4]
```

```
def remove_duplicates(list_1, list_2):  
    list_1_copy = list_1[:]   
    for e in list_1_copy:  
        if e in list_2:  
            list_1.remove(e)
```

```
In [73]: list_1 = [1,2,3,4]
```

```
In [74]: list_2 = [1,2,5,6]
```

```
In [75]: remove_duplicates(list_1, list_2)
```

```
In [76]: print(list_1)  
[3, 4]
```

Problem:

- Python uses an internal counter to iterate through `list_1`.
- If we remove item 1, `list_1` becomes `[2, 3, 4]`. Then the `for` accesses position 1 of the list, which corresponds now to item 3 (so, number 2 is skipped).

Correct implementation:

- The program iterates on the copy of the first list: `list_1_copy`
- The items are deleted from the original `list_1`

Example

Make a program that implements the Sieve of Eratosthenes.

It is an ancient algorithm to search for all prime numbers up to a certain integer (MAX), that in our program will be entered by the user.

Algorithm

1. For each number in the range $[2, \text{MAX}]$:
 - i. If the number is not in list NP (No Primes), then it is prime, and is added to list P (Primes).
 - ii. Add all the multiples of the number up to MAX to list NP .
2. Print list P .

Sieve of Eratosthenes

```
MAX = int(input("Max num: "))
```

```
NP=[]
```

```
P=[]
```

```
for num in range(2,MAX+1):
```

```
    if num not in NP:
```

```
        P.append(num)
```

```
        i=2
```

```
        multiple = num*i
```

```
        while (multiple <=MAX):
```

```
            NP.append(multiple)
```

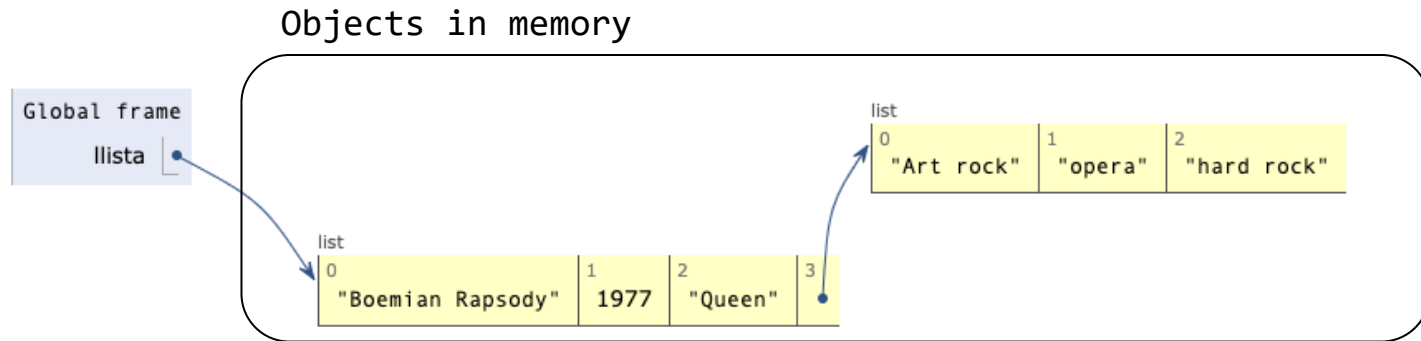
```
            i+=1
```

```
            multiple =num*i
```

```
print(P)
```

Lists in memory

- They are mutable: list is the same object



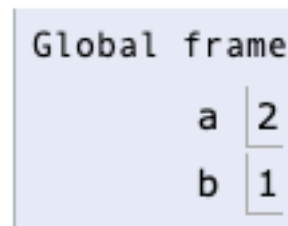
- They behave differently from immutable types

```
In [22]: a=1
```

```
In [23]: b=a
```

```
In [24]: a=2
```

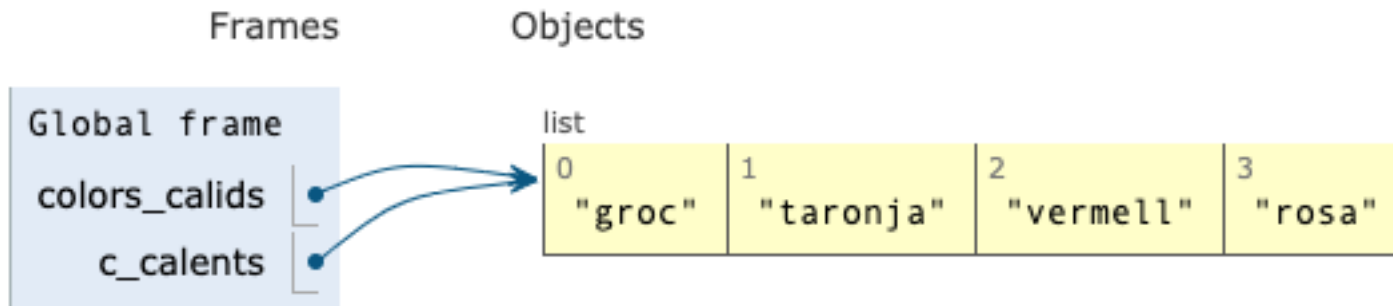
```
In [25]: print(a,b)
2 1
```



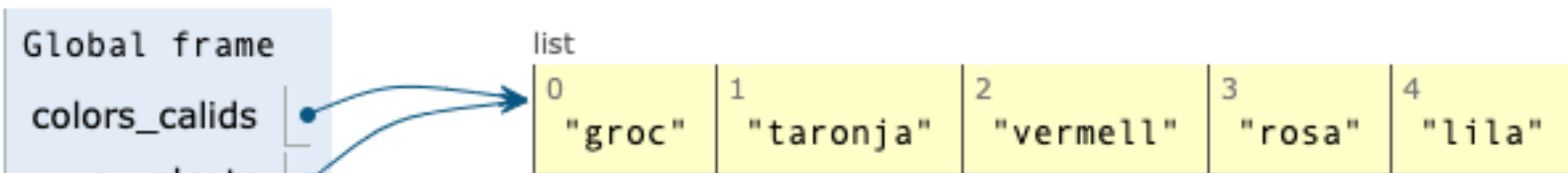
- Lists are objects in memory where the variable name points to the object, so any variable pointing to the object is affected

Analogy: Alias

- Each variable name pointing to the same list is as if it were a synonym:



- One collateral effect is that if we modify one list, we modify the other:



```
In [46]: colors_calids=["groc","taronja","vermell","rosa"]
```

```
In [47]: c_calents=colors_calids
```

```
In [48]: c_calents.append("lila")
```

```
In [49]: print (colors_calids)
['groc', 'taronja', 'vermell', 'rosa', 'lila']
```

Clone a list

- If we want to copy a list to avoid the effect seen above, we need to copy all items using the `[:]` operator or method `copy()`:

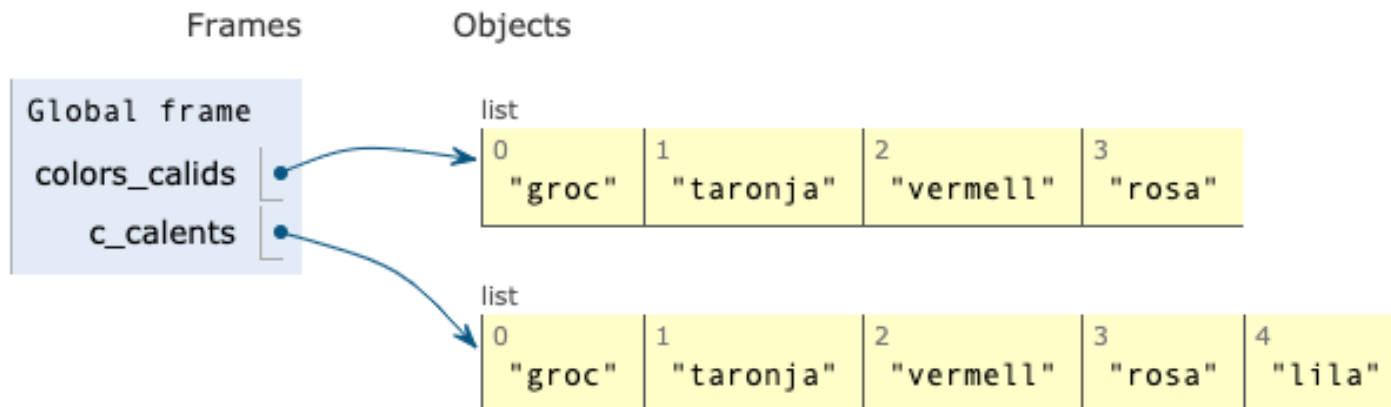
```
In [50]: colors_calids=["groc","taronja","vermell","rosa"]
```

```
In [51]: c_calents = colors_calids[:] ← Alternative:  
c_calents = colors_calids.copy()
```

```
In [52]: print(colors_calids,c_calents)  
['groc', 'taronja', 'vermell', 'rosa'] ['groc', 'taronja', 'vermell',  
'rosa']
```

```
In [53]: c_calents.append("lila")
```

```
In [54]: print(colors_calids,c_calents)  
['groc', 'taronja', 'vermell', 'rosa'] ['groc', 'taronja', 'vermell',  
'rosa', 'lila']
```



Lists of lists of lists ... : collateral effects

- Remember that we can have nested lists.
- Due to the mutability of the lists, we can have unwanted effects. We need to control these collateral effects:

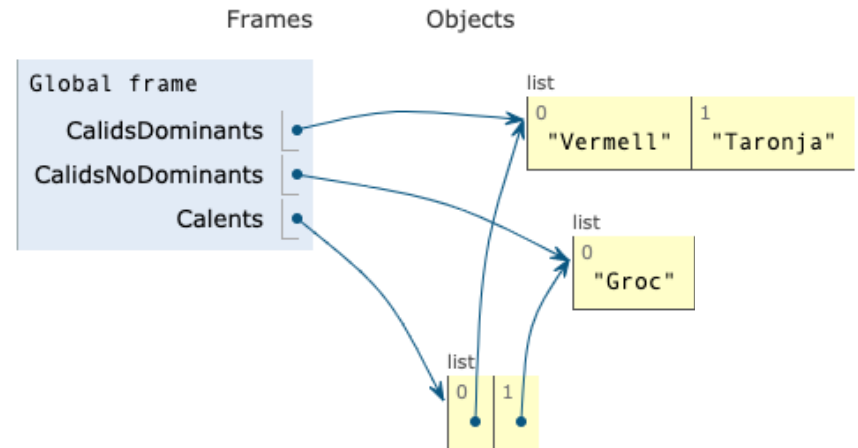
```
In [61]: CalidsDominants =["Vermell","Taronja"]
```

```
In [62]: CalidsNoDominants=["Groc"]
```

```
In [63]: Calents = [CalidsDominants]
```

```
In [64]: Calents.append(CalidsNoDominants)
```

```
In [65]: print(Calents)
[['Vermell', 'Taronja'], ['Groc']]
```



Lists of lists of lists ... : collateral effects

- Remember that we can have nested lists.
- Due to the mutability of the lists, we can have unwanted effects. We need to control these collateral effects:

```
In [61]: CalidsDominants = ["Vermell", "Taronja"]
```

```
In [62]: CalidsNoDominants = ["Groc"]
```

```
In [63]: Calents = [CalidsDominants]
```

```
In [64]: Calents.append(CalidsNoDominants)
```

```
In [65]: print(Calents)
[['Vermell', 'Taronja'], ['Groc']]
```

```
In [66]: CalidsNoDominants.append("Marro")
```

```
In [67]: print(CalidsNoDominants)
['Groc', 'Marro']
```

```
In [68]: print(Calents)
[['Vermell', 'Taronja'], ['Groc', 'Marro']]
```

