04 – Listas en Python 3

Diego Andrés Alvarez Marín
Profesor Asociado
Universidad Nacional de Colombia
Sede Manizales

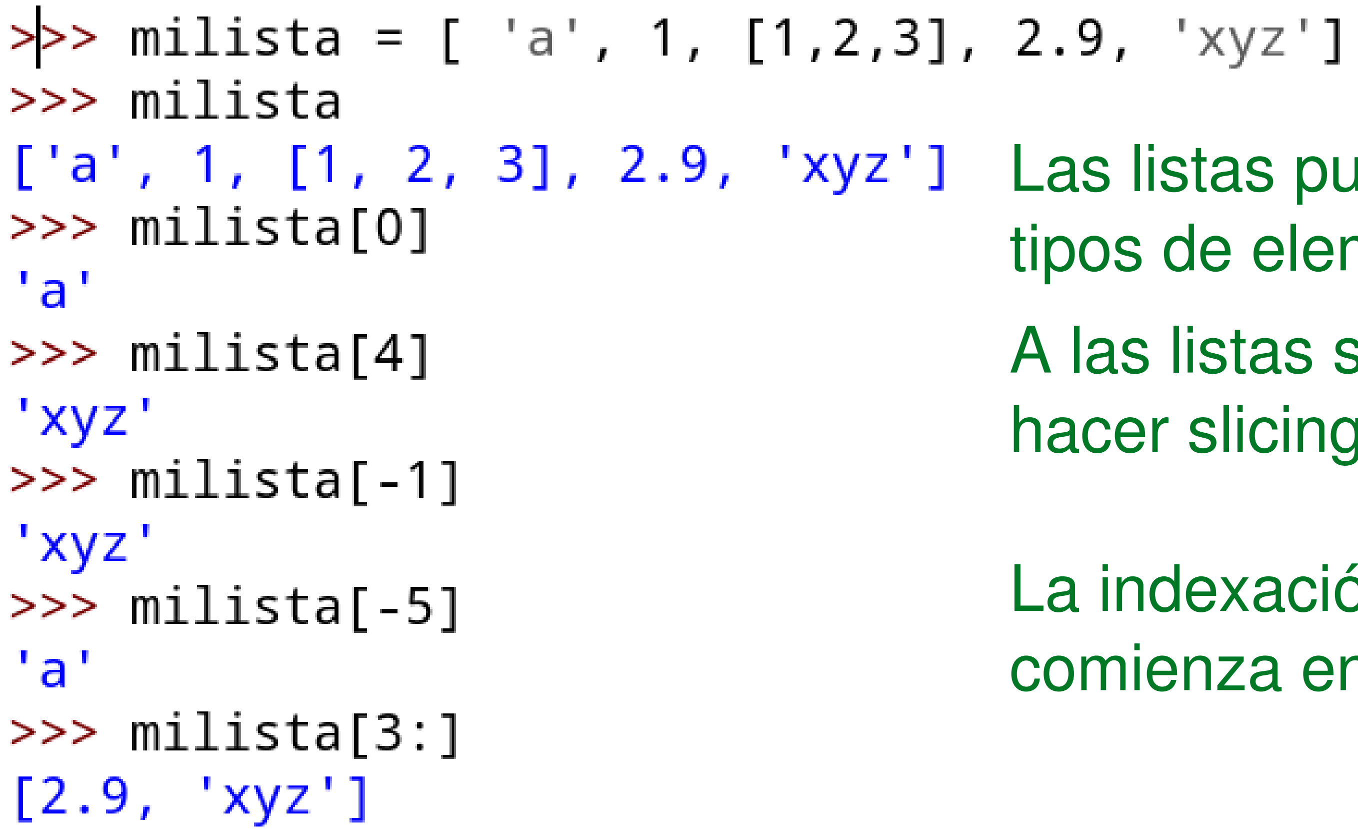
Tabla de contenido

Operadores	**Funciones**:	**Métodos**:
:, +, *	len()	append(), copy()
[:], [i], [i:j], [i:j:k]	id()	extend(), insert()
is, is not	eval()	count(),index()
in, not in	all()	join(), remove()
=, !=	any()	pop(), clear()
<, >, <=, >=	list()	sort(), reverse()
del, []		

Tipos de datos

- Tipos de datos escalares:
  - Números enteros, flotantes, complejos, fracciones, lógicos(booleanos)
- Tipos de datos secuenciales:
  - Secuencias de bytes, cadenas
- Tipos de datos estructurados:
  - Listas (lists): secuencias ordenadas de valores
  - Tuplas (tuples): secuencias inmutables de valores ordenados
  - Conjuntos (sets): conjunto no ordenado de valores
  - Diccionarios (dictionaries): conjunto no ordenado de valores, que tienen una “llave” que los identifican
- Objetos: módulos, funciones, clases, métodos, archivos, código compilado, etc.
- “Constantes”

Listas

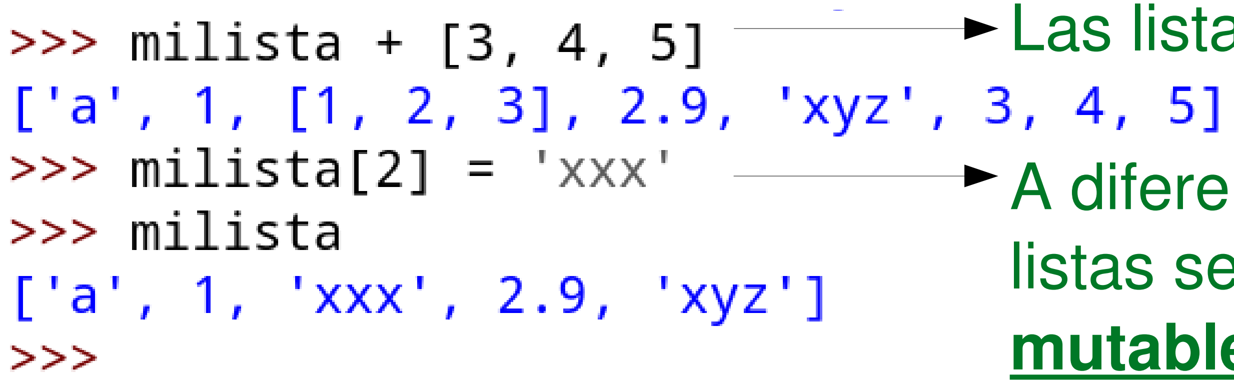


```
>>> milista = [ 'a', 1, [1,2,3], 2.9, 'xyz']
>>> milista
['a', 1, [1, 2, 3], 2.9, 'xyz']
>>> milista[0]
'a'
>>> milista[4]
'xyz'
>>> milista[-1]
'xyz'
>>> milista[-5]
'a'
>>> milista[3:]
[2.9, 'xyz']
```

Las listas pueden tener diferentes tipos de elementos

A las listas se les puede indexar y hacer slicing

La indexación de las listas comienza en 0



```
>>> milista + [3, 4, 5]
['a', 1, [1, 2, 3], 2.9, 'xyz', 3, 4, 5]
>>> milista[2] = 'xxx'
>>> milista
['a', 1, 'xxx', 2.9, 'xyz']
>>>
```

Las listas se pueden concatenar

A diferencia de las cadenas, las listas se pueden modificar: son **mutables**

Listas

```
>>> letras = ['a', 'b', 'c', 'd']
>>> len(letras)
4
>>> vacía = [] # una lista vacía
>>> len(vacía) # longitud de la lista vacía
0
```

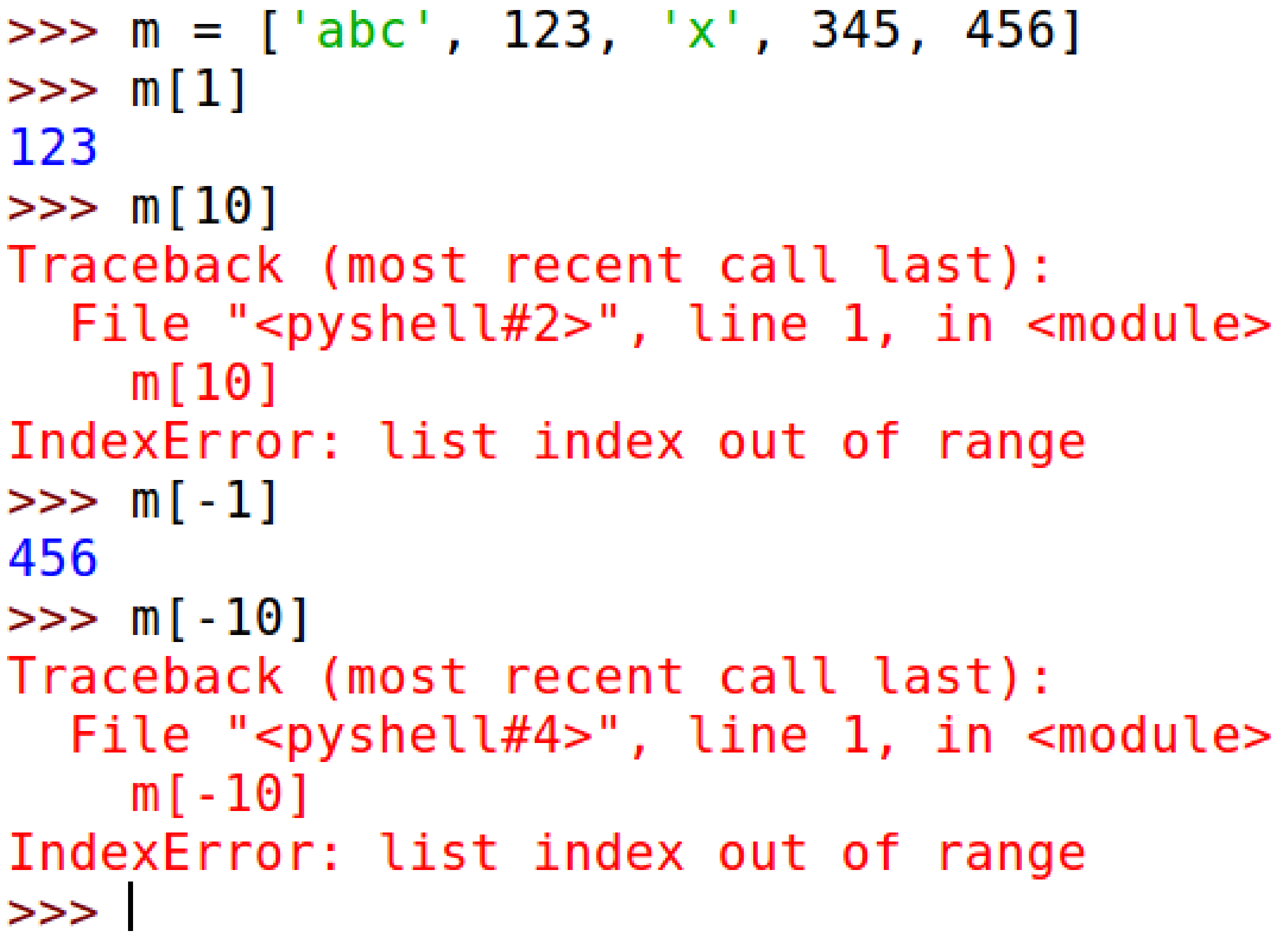
It is possible to nest lists (create lists containing other lists), for example:

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
```

```
>>> x = list(range(10))
>>> x[3:-2]
[3, 4, 5, 6, 7]
>>> x[3:8]
[3, 4, 5, 6, 7]
>>> |
```

```
>>> ['uu', 'vv', 'ww', 'xx', 'yy', 'zz'][2]
'ww'
```

Refiriéndose a elementos fuera de la lista



```
>>> m = ['abc', 123, 'x', 345, 456]
>>> m[1]
123
>>> m[10]
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    m[10]
IndexError: list index out of range
>>> m[-1]
456
>>> m[-10]
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    m[-10]
IndexError: list index out of range
>>> 
```

Las listas pueden contener cualquier tipo de elemento

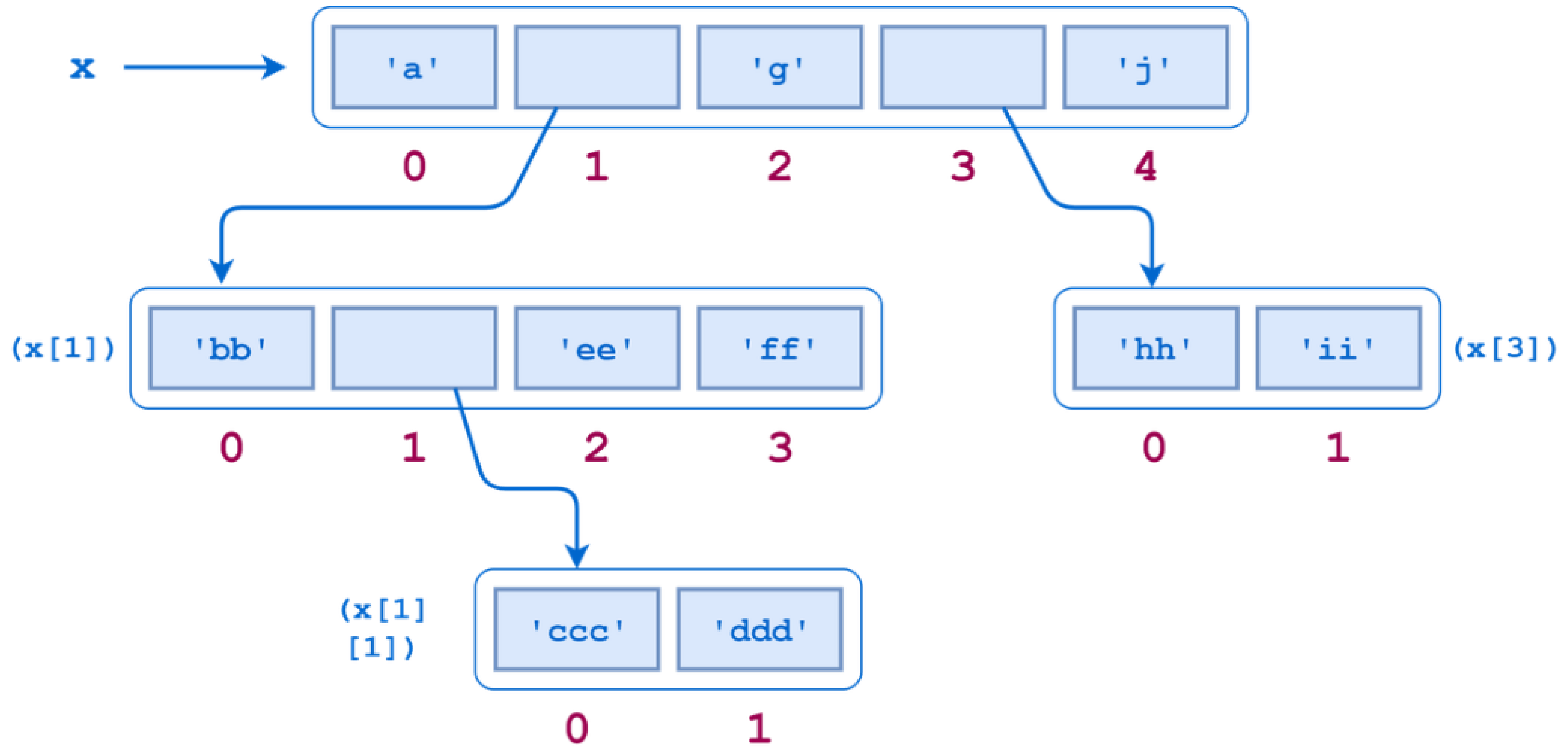
Incluso pueden contener otras listas o incluso funciones, clases, módulos, etc.

```
>>> int
<class 'int'>
>>> len
<built-in function len>
>>> def foo():
...     pass
...
>>> foo
<function foo at 0x035B9030>
>>> import math
>>> math
<module 'math' (built-in)>

>>> a = [int, len, foo, math]
>>> a
[<class 'int'>, <built-in function len>, <function foo at 0x02CA2618>,
<module 'math' (built-in)>]
```

Las listas se pueden anidar

```
>>> x = ['a', ['bb', ['ccc', 'ddd'], 'ee', 'ff'], 'g', ['hh', 'ii'], 'j']
>>> x
['a', ['bb', ['ccc', 'ddd'], 'ee', 'ff'], 'g', ['hh', 'ii'], 'j']
```

The object structure that x references is diagrammed below:



Listas

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> a[1:4]
['bar', 'baz', 'qux']
>>> a[1:4] = [1.1, 2.2, 3.3, 4.4, 5.5]
>>> a
['foo', 1.1, 2.2, 3.3, 4.4, 5.5, 'quux', 'corge']
>>> a[1:6]
[1.1, 2.2, 3.3, 4.4, 5.5]
>>> a[1:6] = ['Bark!']
>>> a
['foo', 'Bark!', 'quux', 'corge']
```

The number of elements inserted need not be equal to the number replaced. Python just grows or shrinks the list as needed.

```
>>> a = [1, 2, 3]
>>> a[1:2] = [2.1, 2.2, 2.3]
>>> a
[1, 2.1, 2.2, 2.3, 3]
```

Note that this is not the same as replacing the single element with a list:

```Python
>>> a = [1, 2, 3]
>>> a[1] = [2.1, 2.2, 2.3]
>>> a
[1, [2.1, 2.2, 2.3], 3]
```

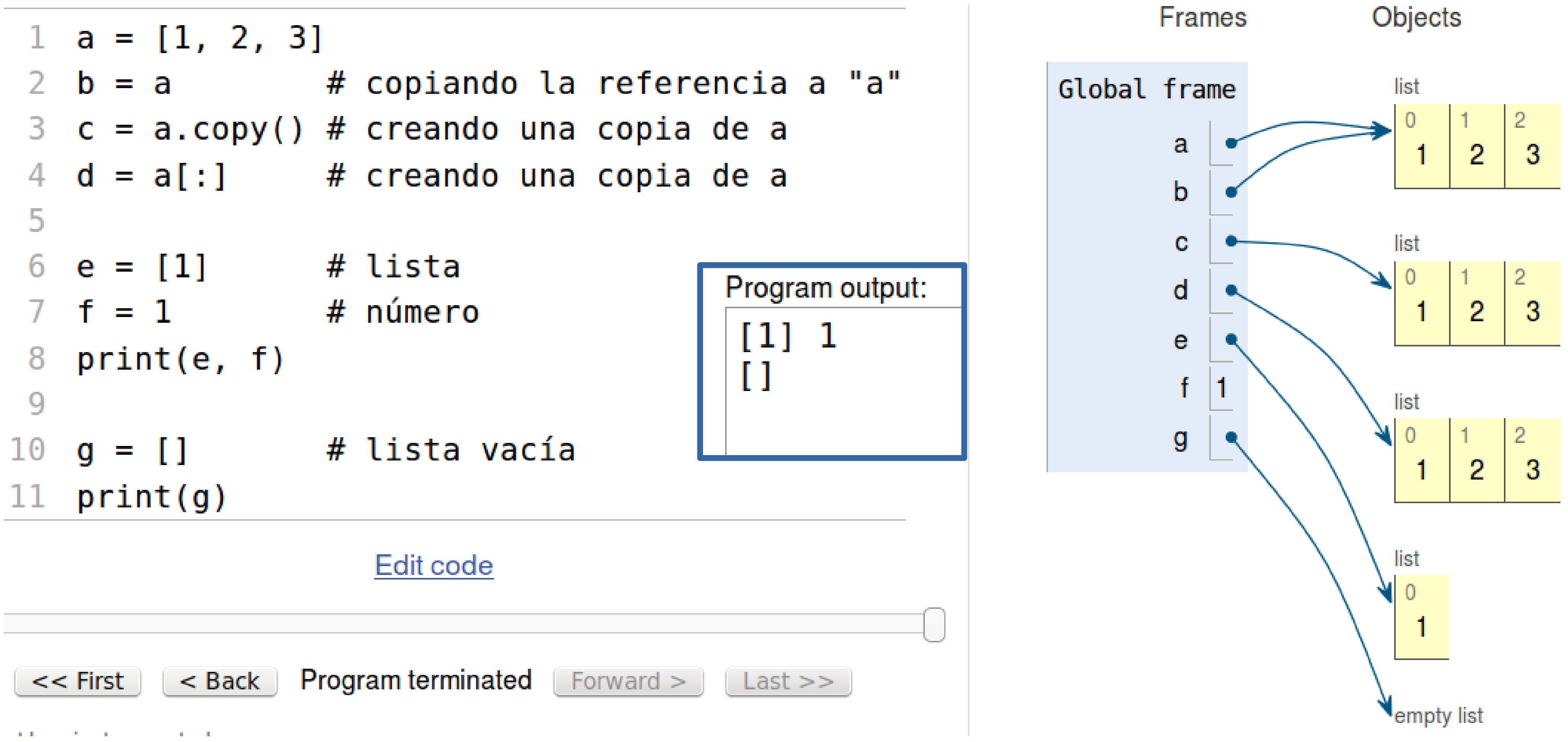
```
>>> a = [1, 2, 7, 8]
>>> a[2:2] = [3, 4, 5, 6]
>>> a
[1, 2, 3, 4, 5, 6, 7, 8]
```

Referencias en Python

```
>>> a = 10
>>> b = a
>>> b = 50
>>> a
10
>>>
>>> c = ['x', 1, 'vc', 2, 3, 4, 5]
>>> d = c
>>> d[2] = 'Hola'
>>> c
['x', 1, 'Hola', 2, 3, 4, 5]
>>> d
['x', 1, 'Hola', 2, 3, 4, 5]
>>> c = ['x', 1, 'vc', 2, 3, 4, 5]
>>> d = c[:]
>>> d[2] = 'Hola'
>>> c
['x', 1, 'vc', 2, 3, 4, 5]
>>> d
['x', 1, 'Hola', 2, 3, 4, 5]
>>>
>>> e = c.copy()
>>> e[2] = 'Hola'
>>> c
['x', 1, 'vc', 2, 3, 4, 5]
>>> e
['x', 1, 'Hola', 2, 3, 4, 5]
```

- Se aplican a tipos de datos mutables (ejemplo: listas y diccionarios).
- El operador = es un operador que copia referencias. La instrucción d=c copia una referencia a lista, no la lista misma
- Utilice slicing[:] o el método copy() si quiere copiar como tal la variable

Visualice la ejecución del código con http://www.pythontutor.com/



```python
a = [1, 2, 3]
b = a          # copiando la referencia a "a"
c = a.copy() # creando una copia de a
d = a[:]       # creando una copia de a

e = [1]        # lista
f = 1          # número
print(e, f)

g = []         # lista vacía
print(g)
```

Program output:

```
[1] 1
[]
```

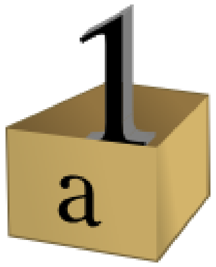
Revisando de nuevo el operador de asignación

=

Other languages have "variables"

In many other languages, assigning to a variable puts a value into a box.

```
int a = 1;
```



Box "a" now contains an integer 1.

Assigning another value to the same variable replaces the contents of the box:

```
a = 2;
```

Now box "a" contains an integer 2.

Assigning one variable to another makes a copy of the value and puts it in the new box:

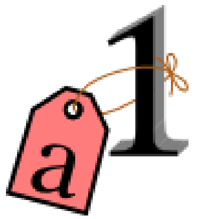
```
int b = a;
```

"b" is a second box, with a copy of integer 2. Box "a" has a separate copy.

Python has "names"

In Python, a "name" or "identifier" is like a parcel tag (or nametag) attached to an object.

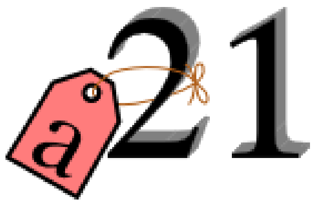
```
a = 1
```



Here, an integer 1 object has a tag labelled "a".

If we reassign to "a", we just move the tag to another object:

```
a = 2
```



Este proceso se llama en Python "garbage collection"

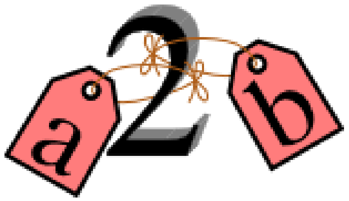
http://en.wikipedia.org/wiki/Garbage_collection_%28computer_science%29

Now the name "a" is attached to an integer 2 object.

The original integer 1 object no longer has a tag "a". It may live on, but we can't get to it through the name "a". (When an object has no more references or tags, it is removed from memory.)

If we assign one name to another, we're just attaching another nametag to an existing object:

```
b = a
```



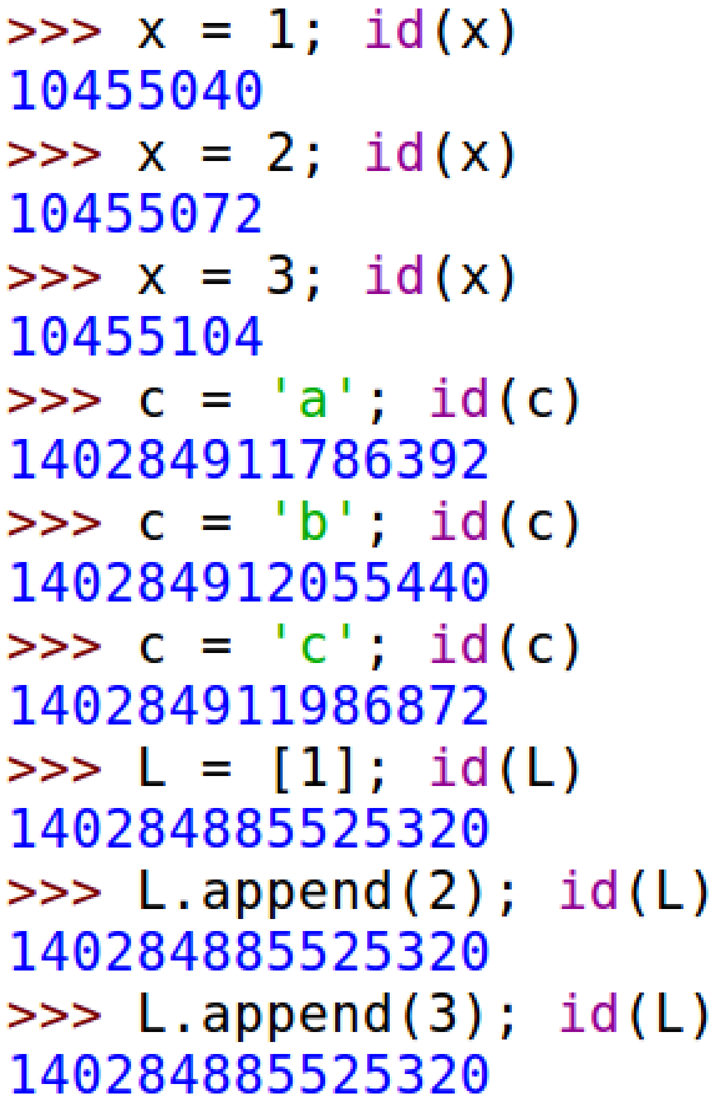
The name "b" is just a second tag bound to the same object as "a".

Although we commonly refer to "variables" even in Python (because it's common terminology), we really mean "names" or "identifiers". In Python, "variables" are nametags for values, not labelled boxes.

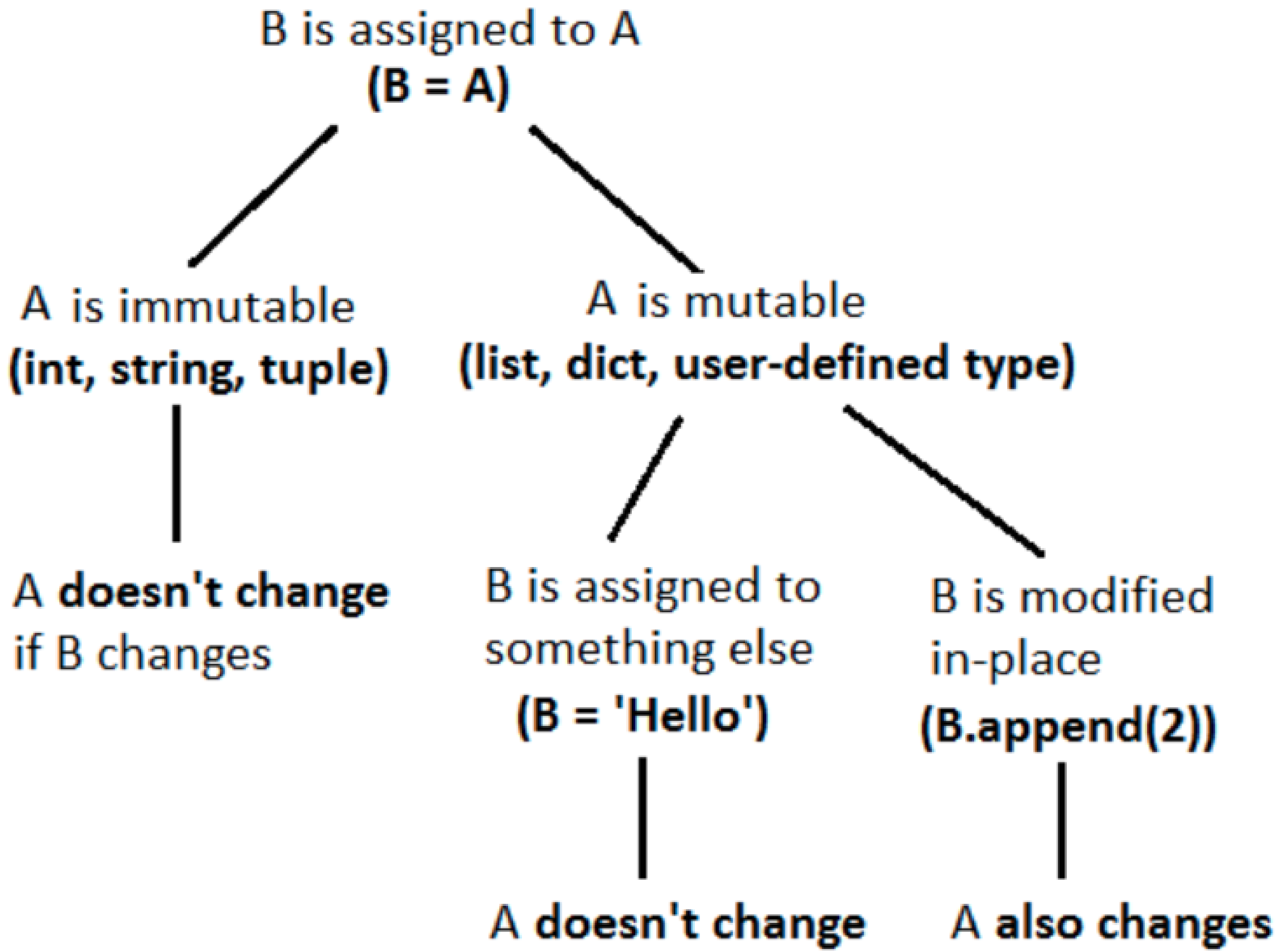
o referencias

Tipos de datos mutables vs inmutables

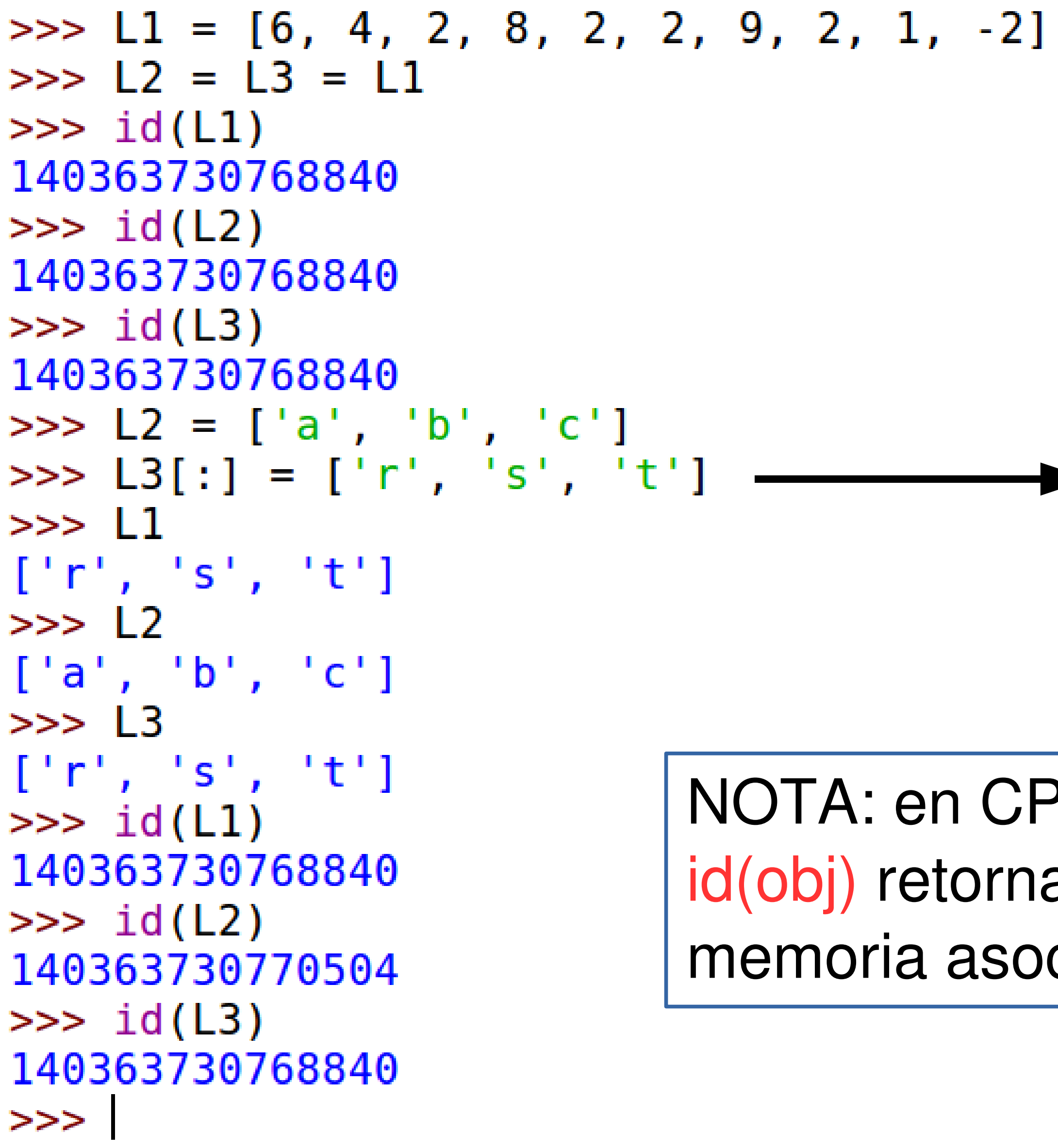
- **INMUTABLES**: An object with a fixed value. Immutable objects include numbers, strings and tuples. Such an object cannot be altered. A new object has to be created if a different value has to be stored. They play an important role in places where a constant hash value is needed, for example as a key in a dictionary: ejemplo: floats, cadenas, tuplas
- **MUTABLES**: Mutable objects can change their value but keep their id(): ejemplo: listas, diccionarios



```
>>> x = 1; id(x)
10455040
>>> x = 2; id(x)
10455072
>>> x = 3; id(x)
10455104
>>> c = 'a'; id(c)
140284911786392
>>> c = 'b'; id(c)
140284912055440
>>> c = 'c'; id(c)
140284911986872
>>> L = [1]; id(L)
140284885525320
>>> L.append(2); id(L)
140284885525320
>>> L.append(3); id(L)
140284885525320
```


B is assigned to A
(B = A)
A is immutable
(int, string, tuple)
A is mutable
(list, dict, user-defined type)
A doesn't change if B changes
B is assigned to something else
(B = 'Hello')
B is modified in-place
(B.append(2))
A doesn't change
A also changes

Modificando una lista conservando la misma ubicación en memoria



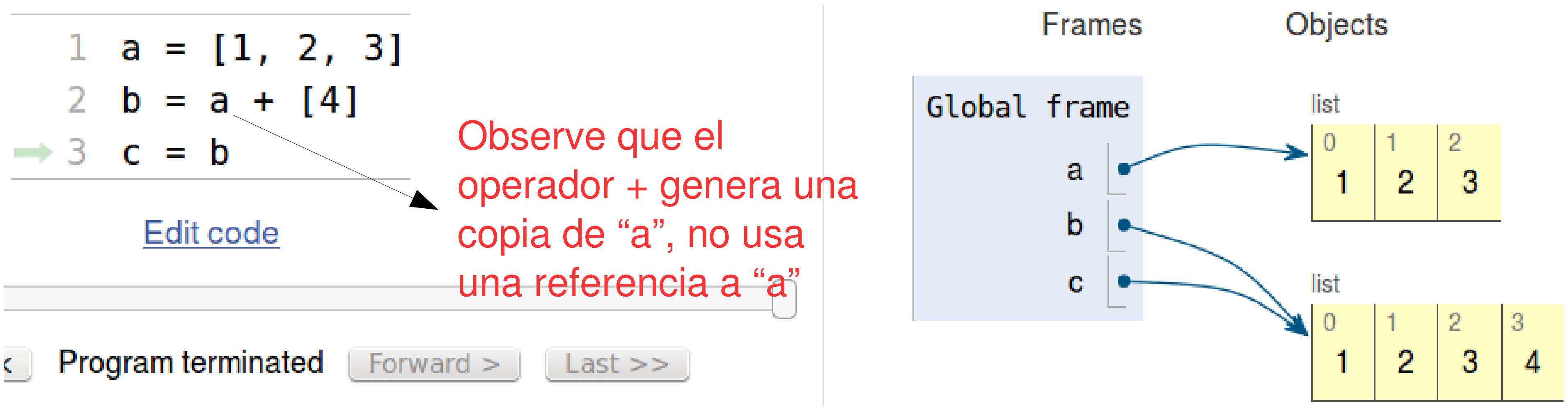
```
>>> L1 = [6, 4, 2, 8, 2, 2, 9, 2, 1, -2]
>>> L2 = L3 = L1
>>> id(L1)
140363730768840
>>> id(L2)
140363730768840
>>> id(L3)
140363730768840
>>> L2 = ['a', 'b', 'c']
>>> L3[:] = ['r', 's', 't']
>>> L1
['r', 's', 't']
>>> L2
['a', 'b', 'c']
>>> L3
['r', 's', 't']
>>> id(L1)
140363730768840
>>> id(L2)
140363730770504
>>> id(L3)
140363730768840
>>> |
```

$\longrightarrow$ observe el uso del comando [:]

NOTA: en CPython, el comando id(obj) retorna la dirección de memoria asociada a obj

Los operadores + y * con listas

```
>>> x = [1, ['x','y', 'z'], 3]
>>> 3*x
[1, ['x', 'y', 'z'], 3, 1, ['x', 'y', 'z'], 3, 1, ['x', 'y', 'z'], 3]
>>> x*3
[1, ['x', 'y', 'z'], 3, 1, ['x', 'y', 'z'], 3, 1, ['x', 'y', 'z'], 3]
>>> y = ['aa', 'bb']
>>> 2*x + y
[1, ['x', 'y', 'z'], 3, 1, ['x', 'y', 'z'], 3, 'aa', 'bb']
>>> |
```



```
a = [1, 2, 3]
b = a + [4]
c = b
```

Observe que el operador + genera una copia de “a”, no usa una referencia a “a”

Comparación de secuencias

Sequence objects may be compared to other objects with the same sequence type. The comparison uses *lexicographical* ordering: first the first two items are compared, and if they differ this determines the outcome of the comparison; if they are equal, the next two items are compared, and so on, until either sequence is exhausted. If two items to be compared are themselves sequences of the same type, the lexicographical comparison is carried out recursively. If all items of two sequences compare equal, the sequences are considered equal. If one sequence is an initial sub-sequence of the other, the shorter sequence is the smaller (lesser) one. Lexicographical ordering for strings uses the Unicode code point number to order individual characters. Some examples of comparisons between sequences of the same type:

```
(1, 2, 3)              < (1, 2, 4)
[1, 2, 3]              < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4)           < (1, 2, 4)
(1, 2)                 < (1, 2, -1)
(1, 2, 3)             == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab'))   < (1, 2, ('abc', 'a'), 4)
```

Note that comparing objects of different types with `<` or `>` is legal provided that the objects have appropriate comparison methods. For example, mixed numeric types are compared according to their numeric value, so 0 equals 0.0, etc. Otherwise, rather than providing an arbitrary ordering, the interpreter will raise a `TypeError` exception.

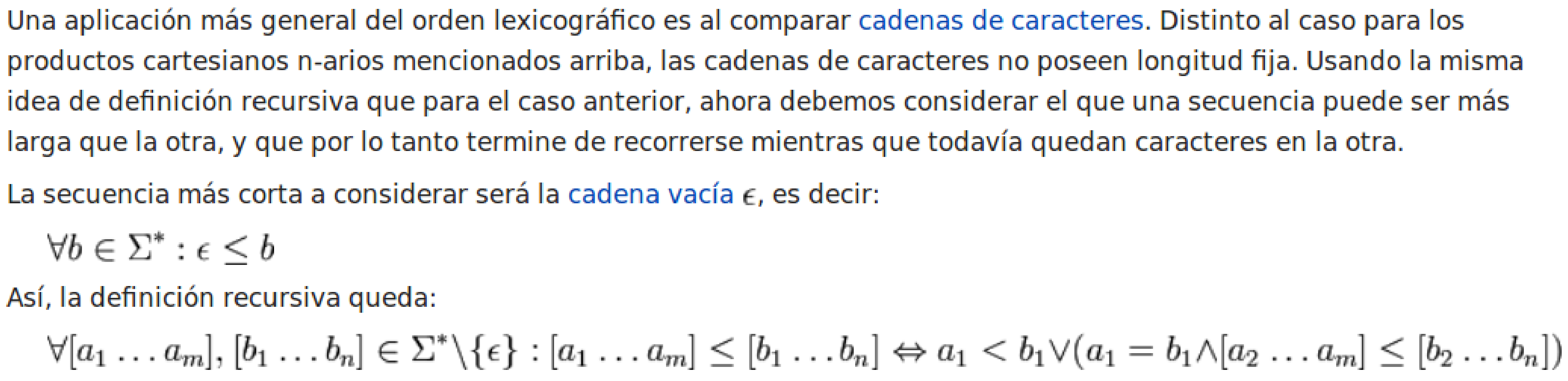
https://docs.python.org/3/tutorial/datastructures.html#comparing-sequences-and-other-types

Ordenamiento lexicográfico de cadenas

(se aplica de igual forma a listas)

The comparison uses **lexicographical ordering**: first the first two items are compared, and if they differ this determines the outcome of the comparison; if they are equal, the next two items are compared, and so on, until either sequence is exhausted. Los diccionarios se ordenan de esta forma.

http://es.wikipedia.org/wiki/Orden_lexicográfico



Una aplicación más general del orden lexicográfico es al comparar cadenas de caracteres. Distinto al caso para los productos cartesianos n-arios mencionados arriba, las cadenas de caracteres no poseen longitud fija. Usando la misma idea de definición recursiva que para el caso anterior, ahora debemos considerar el que una secuencia puede ser más larga que la otra, y que por lo tanto termine de recorrerse mientras que todavía quedan caracteres en la otra.

La secuencia más corta a considerar será la cadena vacía ϵ, es decir:

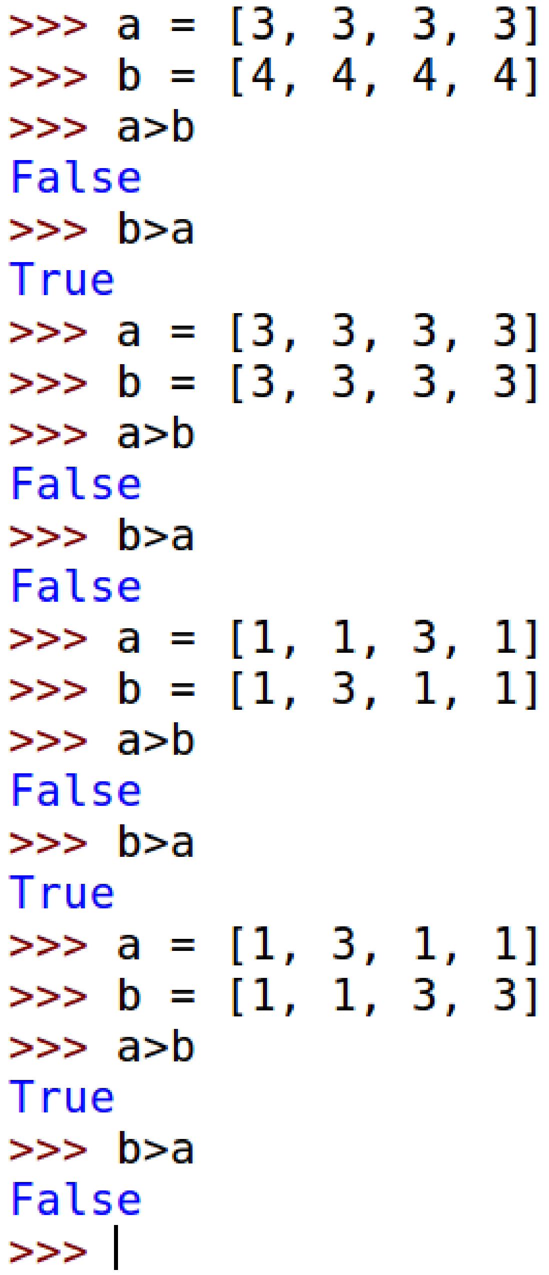
$$\forall b \in \Sigma^* : \epsilon \leq b$$

Así, la definición recursiva queda:

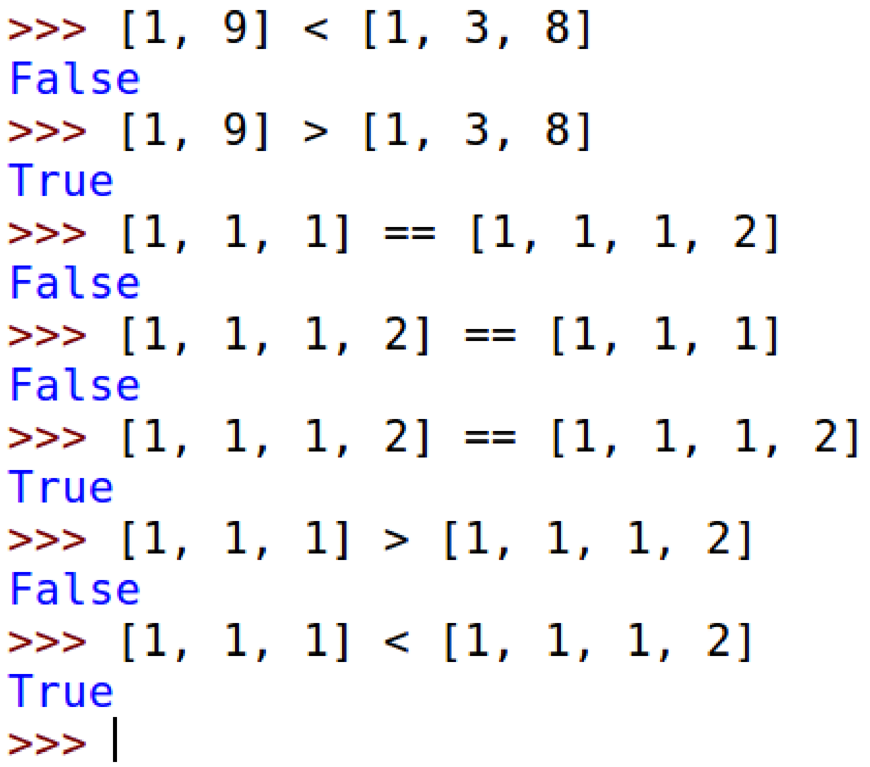
$$\forall [a_1 \ldots a_m], [b_1 \ldots b_n] \in \Sigma^* \setminus \{\epsilon\} : [a_1 \ldots a_m] \leq [b_1 \ldots b_n] \Leftrightarrow a_1 < b_1 \vee (a_1 = b_1 \wedge [a_2 \ldots a_m] \leq [b_2 \ldots b_n])$$

Comparación de listas

The comparison uses **lexicographical ordering**: first the first two items are compared, and if they differ this determines the outcome of the comparison; if they are equal, the next two items are compared, and so on, until either sequence is exhausted. Los diccionarios se ordenan de esta forma.



```
>>> a = [3, 3, 3, 3]
>>> b = [4, 4, 4, 4]
>>> a>b
False
>>> b>a
True
>>> a = [3, 3, 3, 3]
>>> b = [3, 3, 3, 3]
>>> a>b
False
>>> b>a
False
>>> a = [1, 1, 3, 1]
>>> b = [1, 3, 1, 1]
>>> a>b
False
>>> b>a
True
>>> a = [1, 3, 1, 1]
>>> b = [1, 1, 3, 3]
>>> a>b
True
>>> b>a
False
>>> |
```



```
>>> [1, 9] < [1, 3, 8]
False
>>> [1, 9] > [1, 3, 8]
True
>>> [1, 1, 1] == [1, 1, 1, 2]
False
>>> [1, 1, 1, 2] == [1, 1, 1]
False
>>> [1, 1, 1, 2] == [1, 1, 1, 2]
True
>>> [1, 1, 1] > [1, 1, 1, 2]
False
>>> [1, 1, 1] < [1, 1, 1, 2]
True
>>> |
```

Comparación de listas

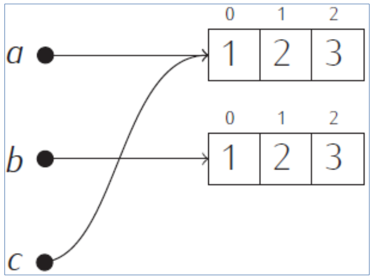
```
If not equal, the sequences are ordered the same as their first
differing elements.  For example, "[1,2,x] <= [1,2,y]" has the same
value as "x <= y".  If the corresponding element does not exist, the
shorter sequence is ordered first (for example, "[1,2] < [1,2,3]").
```

El primer ejemplo falla ya que `type([1,2,3]) !=  type(3)` y el segundo ejemplo es válido ya que `2<3`

```
>>> [1, 2, [1,2,3]] < [1, 2, 3]
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    [1, 2, [1,2,3]] < [1, 2, 3]
TypeError: unorderable types: list() < int()
>>> [1, 2, [1,2,3]] < [1, 3, 3]
True
>>> |
```

Los operadores “is” e “is not”

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> c = a
>>> a == b
True
>>> a is b
False
>>> a == c
True
>>> a is c
True
>>> hex(id(a))
'0xb60ee6cc'
>>> hex(id(b))
'0xb60f34ec'
>>> hex(id(c))
'0xb60ee6cc'
>>>
>>> d = ['x', 'y']
>>> d == ['x', 'y']
True
>>> d is ['x', 'y']
False
```



Los operadores "is" e "is not" verifican la identidad del objeto.

"a is b" retorna True si y solo si “a” y “b” son el mismo objeto: id(a) == id(b)

"a is not b" retorna True si y solo si “a” y “b” no son el mismo objeto: id(a) != id(b)

```
>>> L = [1,2,3]
>>> L == L[:]
True
>>> L is L[:]
False
```

```
>>> a = [1, 2, 1]
>>> a is not [1, 2, 1]
True
```

Comparando “is” e “==”

- `==` y `!=` son operadores que comparan los <u>valores</u> de los objetos
- “`is`” e “`is not`” son operadores que comparan la <u>identidad</u> de los objetos
- Observe que "a == b" no implica que "a is b"
- De igual forma, "a is b" no implica que "a == b"

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a == b
True
>>> a is b
False
```

```
>>> NaN = float('nan')
>>> NaN is NaN
True
>>> NaN == NaN
False
```

El operador is

```
>>> a = [1, 2, 1]
>>> b = [1, 2, 1]
>>> a[0] is b[0]
True
>>> a[1] is b[1]
True
>>> a[2] is b[2]
True
>>> hex(id(a))
'0xb60f34ec'
>>> hex(id(b))
'0xb60f5c2c'
>>> hex(id(a[0]))
'0x83fe500'
>>> hex(id(a[1]))
'0x83fe510'
>>> hex(id(a[2]))
'0x83fe500'
>>> hex(id(b[0]))
'0x83fe500'
>>> hex(id(b[1]))
'0x83fe510'
>>> hex(id(b[2]))
'0x83fe500'
>>> a[0] is b[2]
True
```

Variables “intern”

```
>>> a = [1, 2, 3]
>>> b = [a[0], a[1], a[2]]
>>> a == b
True
>>> a is b
False
>>> b is [b[0], b[1], b[2]]
False
>>> 
```

El comando [:]

En cadenas y tuplas el comando [:] retorna una referencia al mismo objeto.

En el caso de lista se retorna una copia de dicho objeto

[:] no es aplicable a conjuntos y diccionarios

```
In [1]: cadena = 'Hola'

In [2]: cadena[:] is cadena
Out[2]: True

In [3]: tupla = (1,2,3)

In [4]: tupla[:] is tupla
Out[4]: True

In [5]: lista = [1, 2, 3]

In [6]: lista[:] is lista
Out[6]: False

In [7]: conjunto = {1, 2, 3}

In [8]: conjunto[:] is conjunto
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-8-58eba302ae76> in <module>()
----> 1 conjunto[:] is conjunto

TypeError: 'set' object is not subscriptable

In [9]: diccionario = {'nombre': 'Pepito', 'apellido': 'Perez'}

In [10]: diccionario[:] is diccionario
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-10-e9a6f2865d31> in <module>()
----> 1 diccionario[:] is diccionario

TypeError: unhashable type: 'slice'
```

Usando == o is para comparar contra None

Suponiendo que “x = None”, es preferible hacer comparaciones del tipo "if x is None" que "if x == None". Ambas sirven en este caso, ya que solo existe un objeto None. Sin embargo, se prefiere la comparación con "is" ya que el uso de "is" es más rápido y más predecible, ya que este comando solo compara la identidad del objeto, mientras que el comportamiento de == depende del tipo exacto de operandos con los que se está trabajando.

Esta recomendación se hace en:
https://www.python.org/dev/peps/pep-0008/#programming-recommendations

- Comparisons to singletons like None should always be done with `is` or `is not` , never the equality operators.

  Also, beware of writing `if x` when you really mean `if x is not None` -- e.g. when testing whether a variable or argument that defaults to None was set to some other value. The other value might have a type (such as a container) that could be false in a boolean context!

- Use `is not` operator rather than `not ... is` . While both expressions are functionally identical, the former is more readable and preferred.

  Yes:

  ```
  if foo is not None:
  ```

  No:

  ```
  if not foo is None:
  ```

PEP 8: Style Guide for Python Code:
https://www.python.org/dev/peps/pep-0008

Lectura de listas con eval()

```
>>> L = input('Entre una lista = ')
Entre una lista = [1, 2, 3]
>>> L
'[1, 2, 3]'      ───►  Leyó una cadena de texto
>>>
>>> L = list(input('Entre una lista = '))
Entre una lista = [1, 2, 3]
>>> L            ───►  Formó una lista con la cadena de texto
['[', '1', ',', ' ', '2', ',', ' ', '3', ']']
>>>
>>> L = eval(input('Entre una lista = '))
Entre una lista = [1, 2, 3]
>>> L
[1, 2, 3]        ───►  eval() evaluó la cadena de texto como si fuera
                       una expresión válida de Python.
```

Métodos que modifican listas

• L.append(<obj>)	Appends an object to a list.
• L.extend(<iterable>)	Extends a list with the objects from an iterable.
• L.insert(<index>, <obj>)	Inserts an object into a list.
• L.remove(<obj>)	Removes an object from a list.
• L.pop(index=-1)	Removes an element from a list.

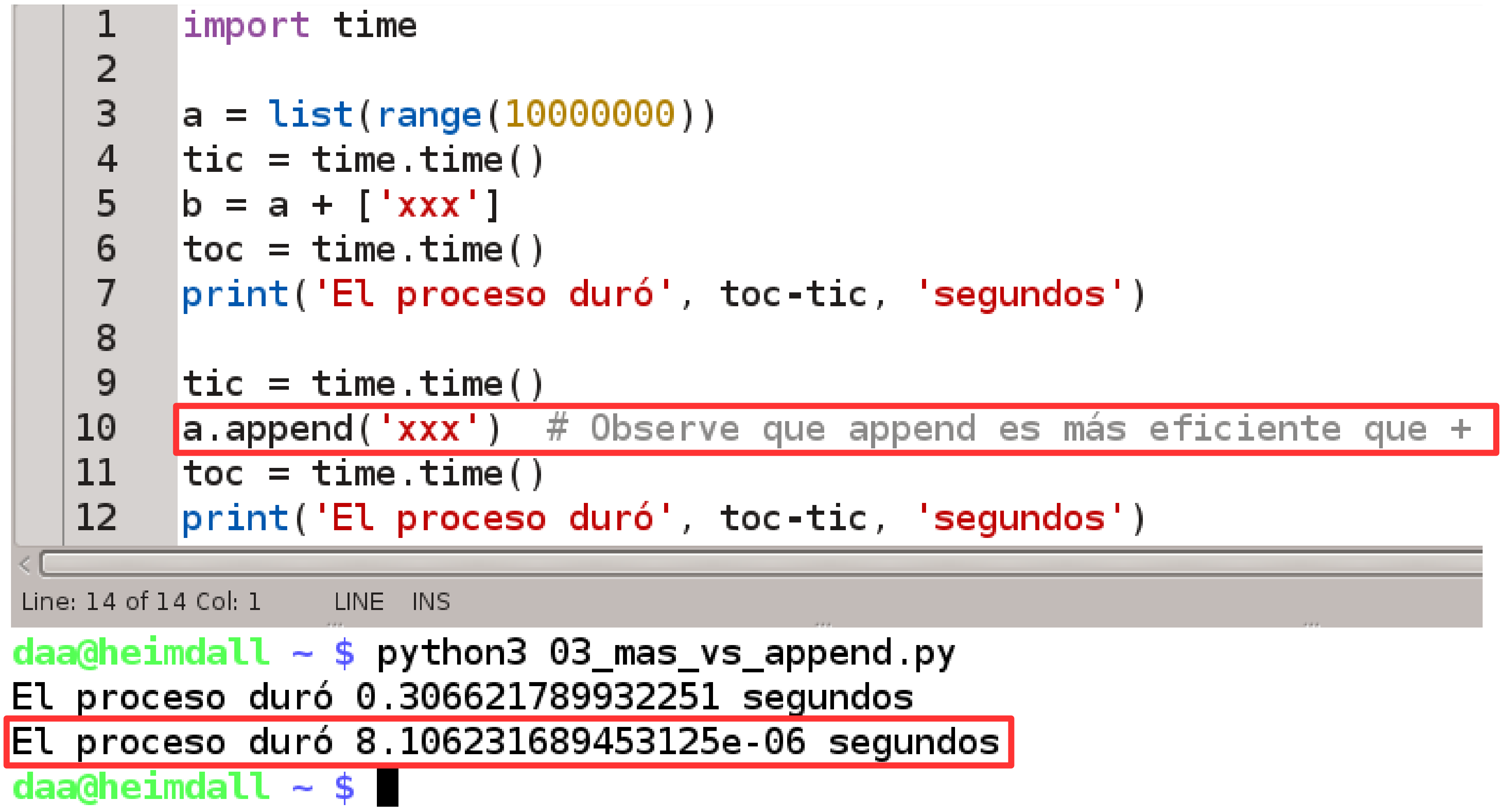
Agregar elementos a una lista, concatenación de listas

El operador + concatena lista con lista, no lista con elemento. Si quiere agregar un elemento a una lista, utilice el método append()

```
>>> L = [1, 2, 3, 4]
>>> S = L + 'x'
Traceback (most recent call last):
  File "<pyshell#82>", line 1, in <module>
    S = L + 'x'
TypeError: can only concatenate list (not "str") to list
>>> S = L + ['x']
>>> R = S
>>> L.append('y')
>>> R.append('z')
>>> L
[1, 2, 3, 4, 'y']
>>> R
[1, 2, 3, 4, 'x', 'z']
>>> S
[1, 2, 3, 4, 'x', 'z']
```

El método append() es mucho más eficiente que utilizar el operador + especialmente con listas de gran tamaño.

+ vs. append()



```
import time

a = list(range(10000000))
tic = time.time()
b = a + ['xxx']
toc = time.time()
print('El proceso duró', toc-tic, 'segundos')

tic = time.time()
a.append('xxx')  # Observe que append es más eficiente que +
toc = time.time()
print('El proceso duró', toc-tic, 'segundos')
```

```
daa@heimdall ~ $ python3 03_mas_vs_append.py
El proceso duró 0.306621789932251 segundos
El proceso duró 8.106231689453125e-06 segundos
daa@heimdall ~ $ 
```

Insertar elementos en una lista

```
>>> a_list = ['a']
>>> a_list = a_list + [2.0, 3]                ①
>>> a_list                                    ②
['a', 2.0, 3]
>>> a_list.append(True)                       ③
>>> a_list
['a', 2.0, 3, True]
>>> a_list.extend(['four', 'Ω'])              ④
>>> a_list
['a', 2.0, 3, True, 'four', 'Ω']
>>> a_list.insert(0, 'Ω')                     ⑤
>>> a_list
['Ω', 'a', 2.0, 3, True, 'four', 'Ω']
```

No use la forma 1 con listas muy grandes, ya que es ineficiente.

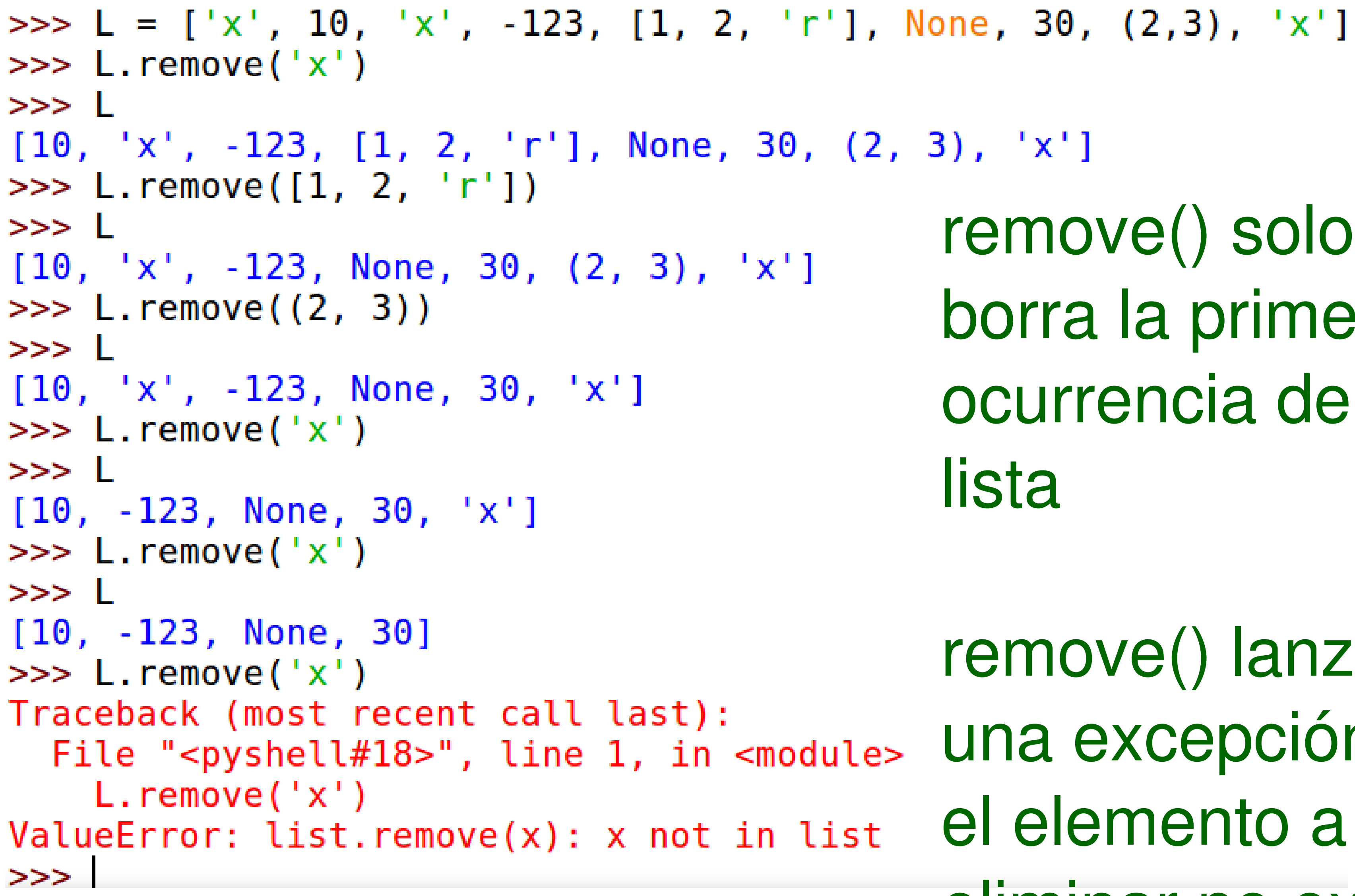
2. append() agrega un elemento a la lista

4. extend() puede agregar varios elementos a la lista

5. Con insert() se pueden poner elementos en cualquier posición de la lista

```
>>> a_list = ['a', 'b', 'c']
>>> a_list.extend(['d', 'e', 'f'])  ①
>>> a_list
['a', 'b', 'c', 'd', 'e', 'f']
>>> len(a_list)                     ②
6
>>> a_list[-1]
'f'
>>> a_list.append(['g', 'h', 'i'])  ③
>>> a_list
['a', 'b', 'c', 'd', 'e', 'f', ['g', 'h', 'i']]
>>> len(a_list)                     ④
7
>>> a_list[-1]
['g', 'h', 'i']
```

Eliminar elementos de una lista



```
>>> L = ['x', 10, 'x', -123, [1, 2, 'r'], None, 30, (2,3), 'x']
>>> L.remove('x')
>>> L
[10, 'x', -123, [1, 2, 'r'], None, 30, (2, 3), 'x']
>>> L.remove([1, 2, 'r'])
>>> L
[10, 'x', -123, None, 30, (2, 3), 'x']
>>> L.remove((2, 3))
>>> L
[10, 'x', -123, None, 30, 'x']
>>> L.remove('x')
>>> L
[10, -123, None, 30, 'x']
>>> L.remove('x')
>>> L
[10, -123, None, 30]
>>> L.remove('x')
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    L.remove('x')
ValueError: list.remove(x): x not in list
>>> 
```

remove() solo borra la primera ocurrencia de la lista

remove() lanzará una excepción si el elemento a eliminar no existe.

Eliminando elementos de una lista con pop()

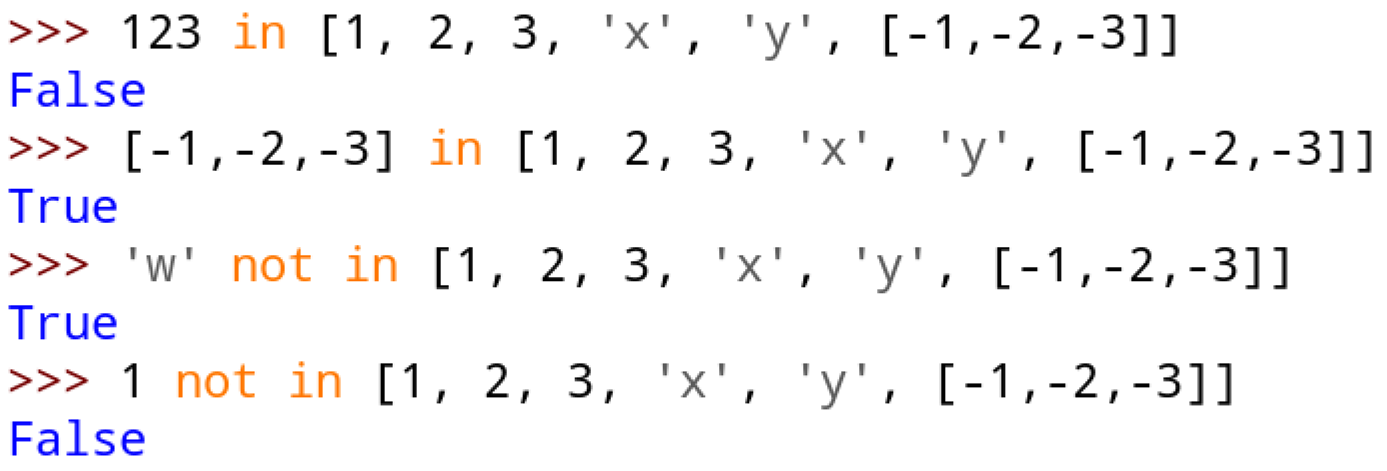
```
>>> L = [1, 'x', [1,2,3], 'abc']
>>> elem = L.pop()
>>> elem
'abc'
>>> L
[1, 'x', [1, 2, 3]]
>>> elem = L.pop(1)
>>> elem
'x'
>>> L
[1, [1, 2, 3]]
>>> L.pop()
[1, 2, 3]
>>> L.pop()
1
>>> L.pop()
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    L.pop()
IndexError: pop from empty list
>>> |
```

pop() con un índice retorna y elimina el elemento indicado

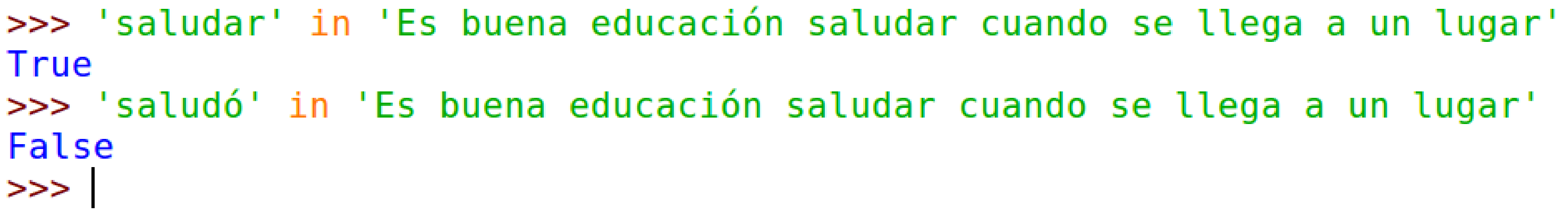
Observe que pop() retorna el elemento eliminado, mientras que remove() y del no lo hacen.

Los operadores `in` y `not in`

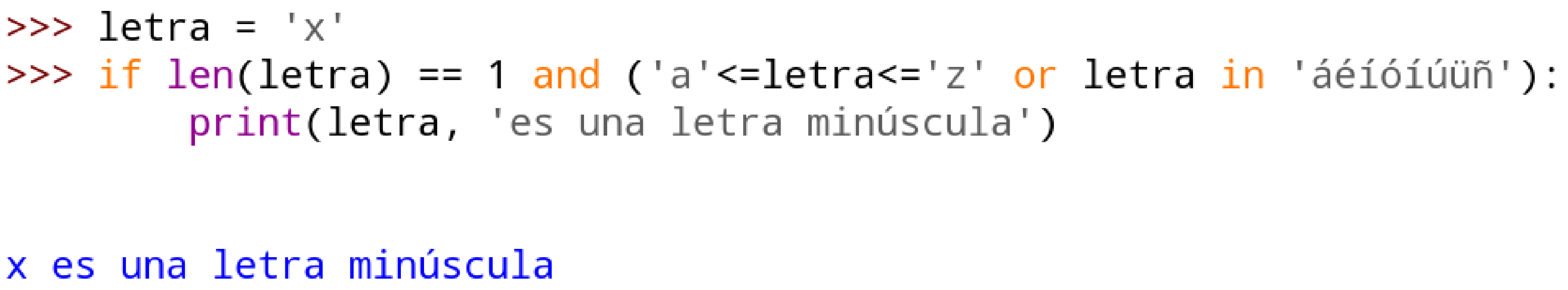
Los operadores `in` y `not in` verifican si un valor está presente en una lista, o si una cadena es una subcadena de otra cadena



```
>>> 123 in [1, 2, 3, 'x', 'y', [-1,-2,-3]]
False
>>> [-1,-2,-3] in [1, 2, 3, 'x', 'y', [-1,-2,-3]]
True
>>> 'w' not in [1, 2, 3, 'x', 'y', [-1,-2,-3]]
True
>>> 1 not in [1, 2, 3, 'x', 'y', [-1,-2,-3]]
False
```



```
>>> 'saludar' in 'Es buena educación saludar cuando se llega a un lugar'
True
>>> 'saludó' in 'Es buena educación saludar cuando se llega a un lugar'
False
>>> |
```



```
>>> letra = 'x'
>>> if len(letra) == 1 and ('a'<=letra<='z' or letra in 'áéíóíúüñ'):
        print(letra, 'es una letra minúscula')

x es una letra minúscula
```

Los operadores `in` y `not in`

Esos operadores se pueden encadenar:

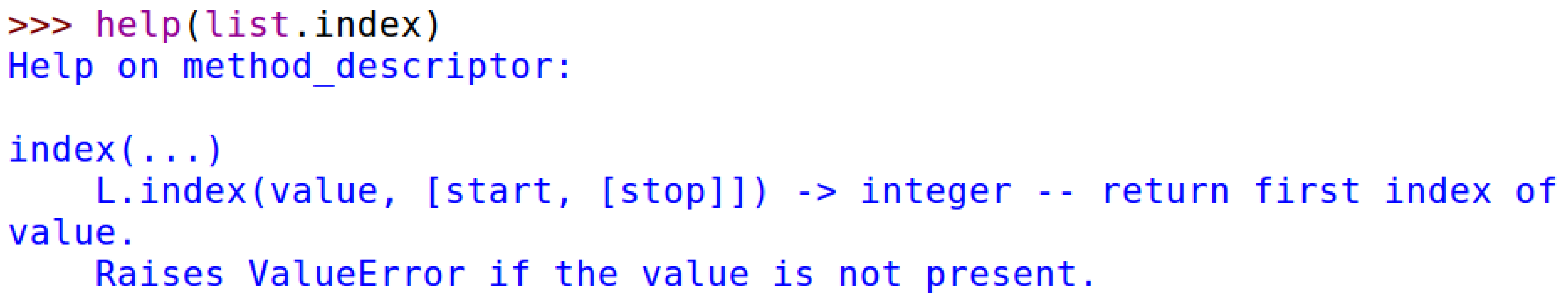
```
>>> 1 in [1,2,3] in [2, 'x', [1,2,3]]
True
>>> 1 in [1,2,3] in [2, 'x', [1,2,4]]
False
>>> 1 in [1,2,3] not in [2, 'x', [1,2,4]]
True
>>> 12 in [1,2,3] not in [2, 'x', [1,2,4]]
False
>>> 12 not in [1,2,3] not in [2, 'x', [1,2,4]]
True
>>> 'o' in 'otro' in 'otro día'
True
>>> 'a' in 'otro' in 'otro día'
False
>>> 'a' not in 'otro' in 'otro día'
True
>>> -1 < 1 in [1,2,3] in [2, 'x', [1,2,3]]
True
```

Buscando valores en una lista

```
>>> a_list = ['a', 'b', 'new', 'mpilgrim', 'new']
>>> a_list.count('new')            ①
2
>>> 'new' in a_list                ②
True
>>> 'c' in a_list
False
>>> a_list.index('mpilgrim')       ③
3
>>> a_list.index('new')            ④
2
>>> a_list.index('c')              ⑤
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: list.index(x): x not in list
```

- 1. count() retorna el número de ocurrencias en una lista
- 2. el operador in es más rápido que if count(L) >= 1:
- 4. index() retorna el índice de la primera ocurrencia
- 5. si index() no encuentra, se lanza una excepción

El método index() en listas



```
>>> help(list.index)
Help on method_descriptor:

index(...)
    L.index(value, [start, [stop]]) -> integer -- return first index of
value.
    Raises ValueError if the value is not present.
```

index() retorna el valor de la primera ocurrencia

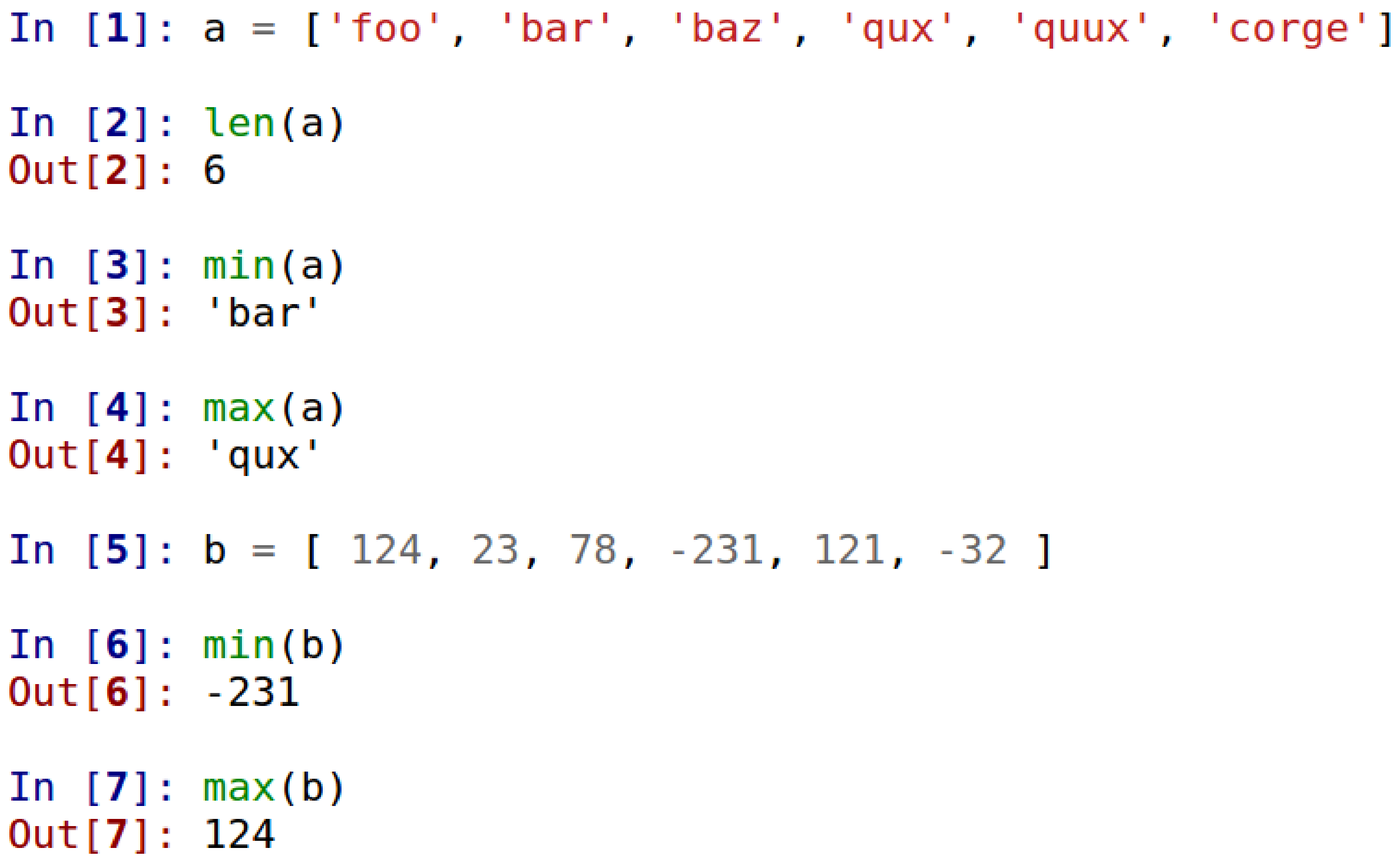
index() puede tener dos argumentos extra para hacer la búsqueda más ágil. El primero dice donde empezar y el último donde terminar (rango del slicing start:stop). Recuerde que el primer índice de la lista es el cero.

El método index() en listas

```
>>> L = [1, 2, 'x', 3, 4, 'x', 5, 6, 7, 'x']
>>> L.index('x')
2
>>> L.index('x',3)
5
>>> L.index('x',6,9)
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    L.index('x',6,9)
ValueError: 'x' is not in list
>>> L.index('x',6)
9
>>> L.index('x',6,20)
9
>>> help(list.index)
Help on method_descriptor:

index(...)
    L.index(value, [start, [stop]]) -> integer -- return first index of value.
    Raises ValueError if the value is not present.
```

El rango analizado corresponde al slicing 6:9

len(), min() y max()



```
In [1]: a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

In [2]: len(a)
Out[2]: 6

In [3]: min(a)
Out[3]: 'bar'

In [4]: max(a)
Out[4]: 'qux'

In [5]: b = [ 124, 23, 78, -231, 121, -32 ]

In [6]: min(b)
Out[6]: -231

In [7]: max(b)
Out[7]: 124
```

Eliminar elementos de una lista

```
>>> milista = [ 'x', 2.3, 1.1, None, [1,2,3], 123.4, -42, 'abw']
>>> del milista[3]
>>> milista
['x', 2.3, 1.1, [1, 2, 3], 123.4, -42, 'abw']
>>> del milista[2:4]
>>> milista
['x', 2.3, 123.4, -42, 'abw']
>>> del milista[:]
>>> milista
[]
>>> del milista
>>> milista
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    milista
NameError: name 'milista' is not defined
>>> |
```

Observe que las listas no quedan con vacíos después de eliminar elementos

Equivalente a `milista.clear()`

Borra la variable milista de la memoria RAM

```
>>> a = [1, 2, 3]
>>> b = a
>>> del a[1]
>>> a
[1, 3]
>>> b
[1, 3]
```

"del" no es aplicable a cadenas (son inmutables)

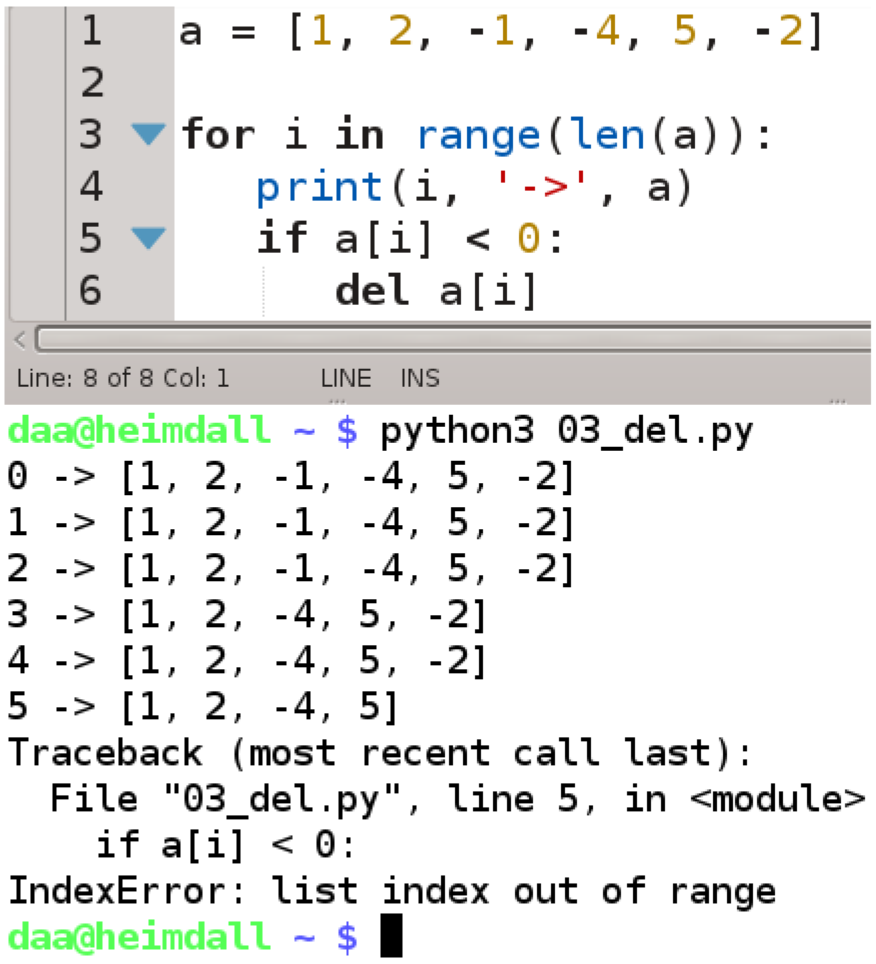
```
>>> a = 'Una cadena'
>>> del a[2]
Traceback (most recent call last):
  File "<pyshell#20>", line 1, in <module>
    del a[2]
TypeError: 'str' object doesn't support item deletion
```

Borrando elementos de una lista con [] y slicing y el método clear() (en vez de usar del)

```
>>> milista = ['x', 2.3, 1.1, None, [1,2,3], 123.4, -42, 'abw']
>>> milista[3] = []   # [] no funciona con indexado
>>> milista
['x', 2.3, 1.1, [], [1, 2, 3], 123.4, -42, 'abw']
>>> milista[2:4] = [] # Es equivalente a "del milista[2:4]"
>>> milista
['x', 2.3, [1, 2, 3], 123.4, -42, 'abw']
>>> milista[:] = []   # Es equivalente a del milista[:]
>>> milista
[]
>>> a = [1, 2, 3]
>>> b = a
>>> a[1] = []
>>> a
[1, [], 3]
>>> b
[1, [], 3]
```

```
>>> L = [1,2,3]
>>> L.clear()
>>> L
[]
```

Error en el uso de “del”



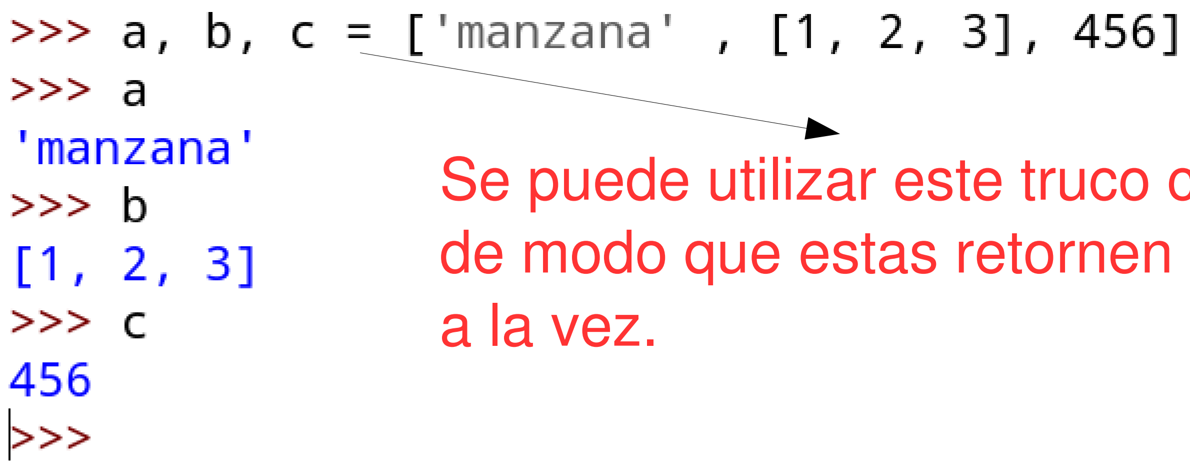
```python
a = [1, 2, -1, -4, 5, -2]

for i in range(len(a)):
    print(i, '->', a)
    if a[i] < 0:
        del a[i]
```

Line: 8 of 8 Col: 1   LINE   INS

```
daa@heimdall ~ $ python3 03_del.py
0 -> [1, 2, -1, -4, 5, -2]
1 -> [1, 2, -1, -4, 5, -2]
2 -> [1, 2, -1, -4, 5, -2]
3 -> [1, 2, -4, 5, -2]
4 -> [1, 2, -4, 5, -2]
5 -> [1, 2, -4, 5]
Traceback (most recent call last):
  File "03_del.py", line 5, in <module>
    if a[i] < 0:
IndexError: list index out of range
daa@heimdall ~ $ 
```

Es una mala práctica de programación eliminar los elementos de una lista sobre la cual se está iterando, ya que se presta para confusiones y errores.

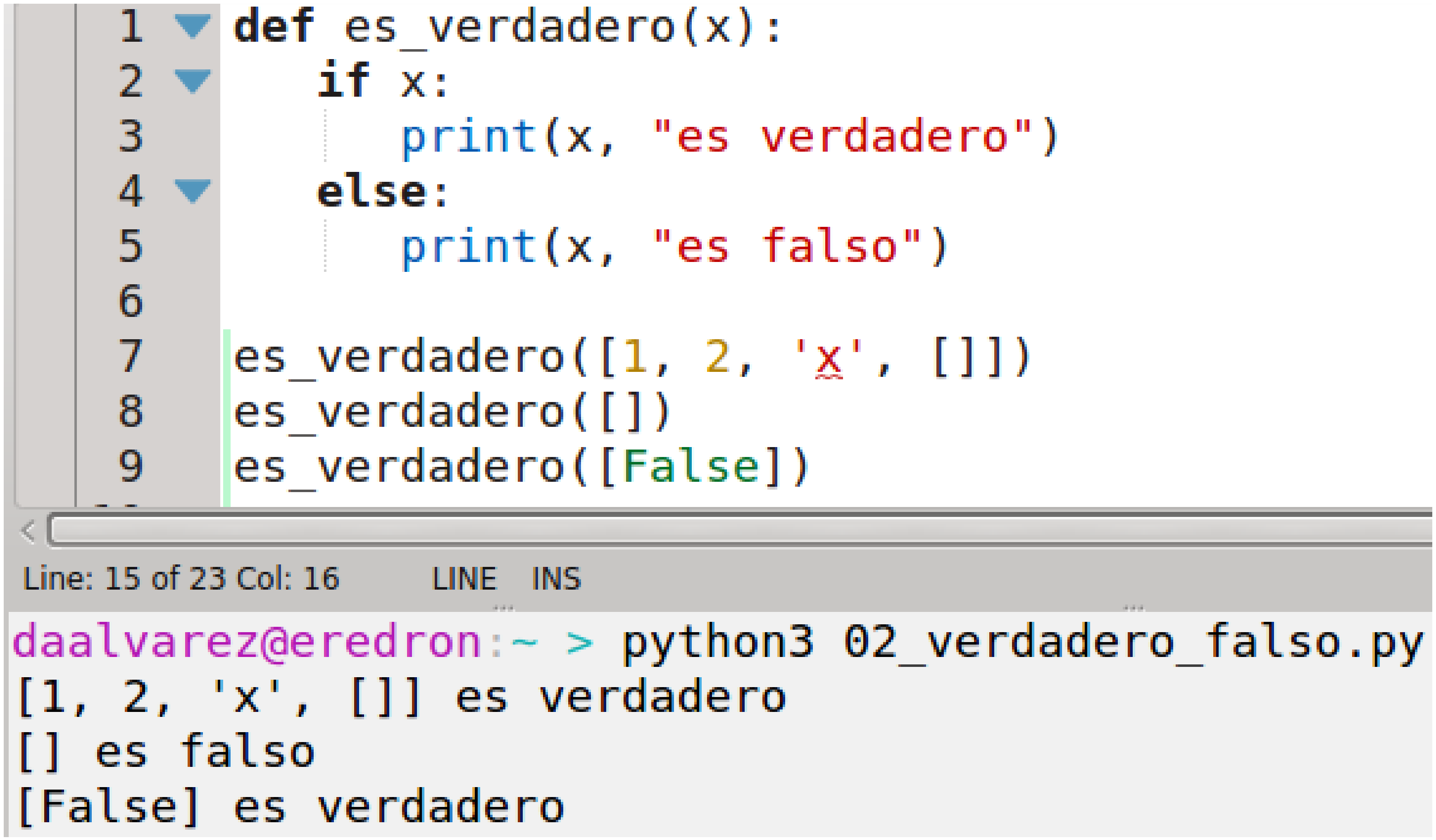
Asignación de elementos usando listas (sequence unpacking)



```
>>> a, b, c = ['manzana' , [1, 2, 3], 456]
>>> a
'manzana'
>>> b
[1, 2, 3]
>>> c
456
>>>
```

Se puede utilizar este truco con funciones, de modo que estas retornen varios valores a la vez.

Listas en un contexto booleano



```
def es_verdadero(x):
    if x:
        print(x, "es verdadero")
    else:
        print(x, "es falso")

es_verdadero([1, 2, 'x', []])
es_verdadero([])
es_verdadero([False])
```

Line: 15 of 23 Col: 16 LINE INS

```
daalvarez@eredron:~ > python3 02_verdadero_falso.py
[1, 2, 'x', []] es verdadero
[] es falso
[False] es verdadero
```

Una lista vacía retorna falso, una lista con al menos un elemento retorna verdadero `list.`**`append`**(*x*)

Add an item to the end of the list. Equivalent to `a[len(a):] = [x]`.

`list.`**`extend`**(*L*)

Extend the list by appending all the items in the given list. Equivalent to `a[len(a):] = L`.

`list.`**`insert`**(*i*, *x*)

Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.

`list.`**`remove`**(*x*)

Remove the first item from the list whose value is *x*. It is an error if there is no such item.

`list.`**`pop`**([*i*])

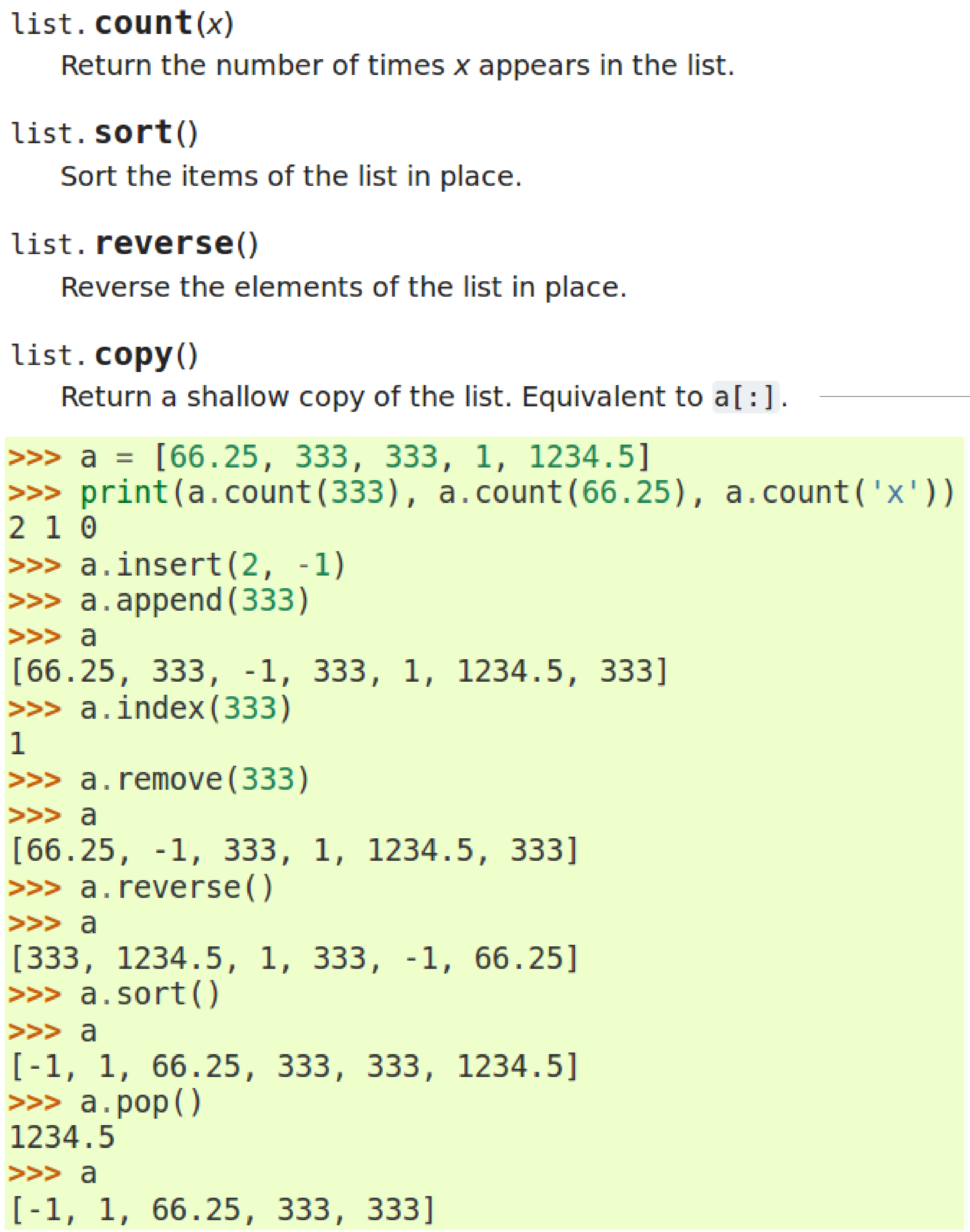
Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. (The square brackets around the *i* in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)

`list.`**`clear`**()

Remove all items from the list. Equivalent to `del a[:]`.

`list.`**`index`**(*x*)

Return the index in the list of the first item whose value is *x*. It is an error if there is no such item.



`list.`**`count`**`(x)`

Return the number of times *x* appears in the list.

`list.`**`sort`**`()`

Sort the items of the list in place.

`list.`**`reverse`**`()`

Reverse the elements of the list in place.

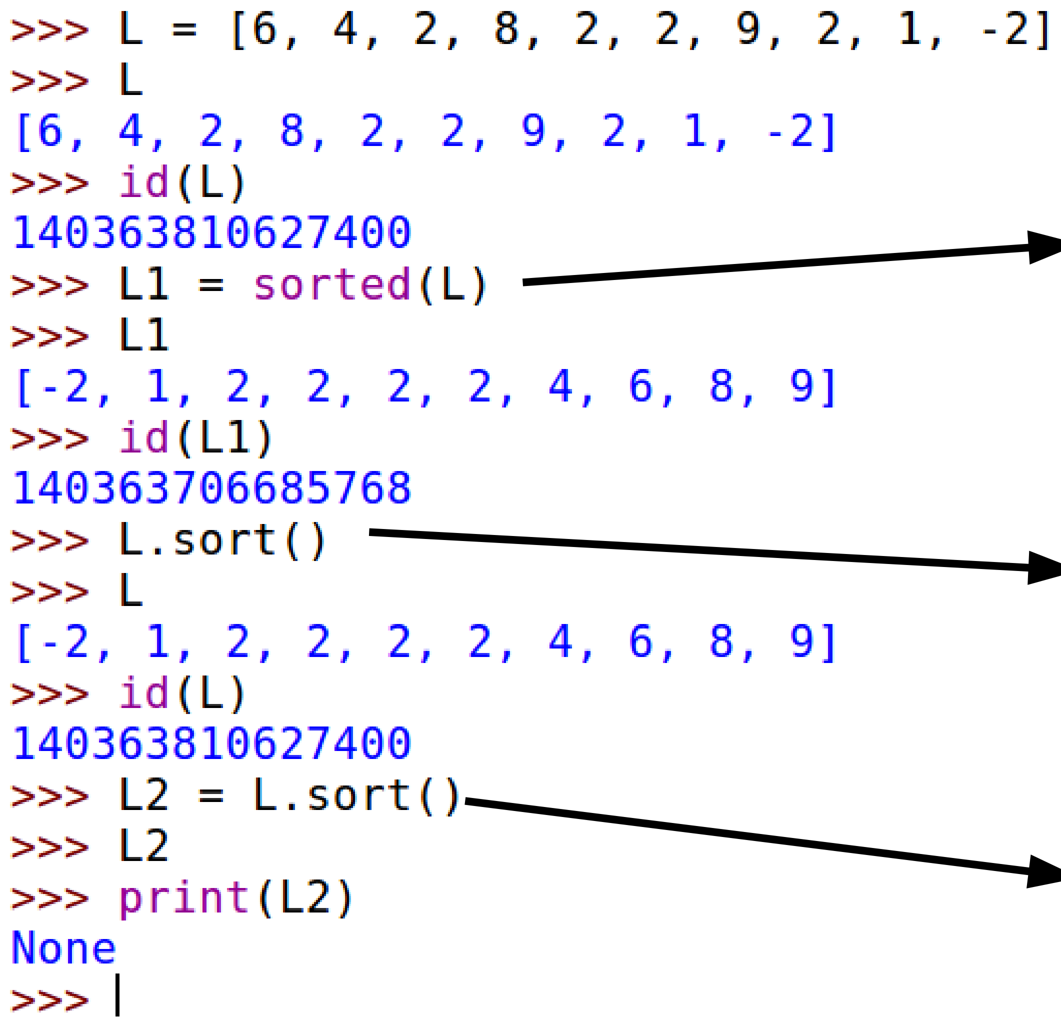
`list.`**`copy`**`()`

Return a shallow copy of the list. Equivalent to `a[:]`. → OJO: “shallow” copy

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print(a.count(333), a.count(66.25), a.count('x'))
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
>>> a.pop()
1234.5
>>> a
[-1, 1, 66.25, 333, 333]
```

insert() ,remove() y sort()
retornan “None”

L.sort() vs sorted(L)



```
>>> L = [6, 4, 2, 8, 2, 2, 9, 2, 1, -2]
>>> L
[6, 4, 2, 8, 2, 2, 9, 2, 1, -2]
>>> id(L)
140363810627400
>>> L1 = sorted(L)
>>> L1
[-2, 1, 2, 2, 2, 2, 4, 6, 8, 9]
>>> id(L1)
140363706685768
>>> L.sort()
>>> L
[-2, 1, 2, 2, 2, 2, 4, 6, 8, 9]
>>> id(L)
140363810627400
>>> L2 = L.sort()
>>> L2
>>> print(L2)
None
>>> |
```

sorted(L) crea una copia

L.sort() no crea una copia

L.sort() retorna None

`all`(*iterable*)

Return `True` if all elements of the *iterable* are true (or if the iterable is empty). Equivalent to:

```python
def all(iterable):
    for element in iterable:
        if not element:
            return False
    return True
```

`any`(*iterable*)

Return `True` if any element of the *iterable* is true. If the iterable is empty, return `False`. Equivalent to:

```python
def any(iterable):
    for element in iterable:
        if element:
            return True
    return False
```

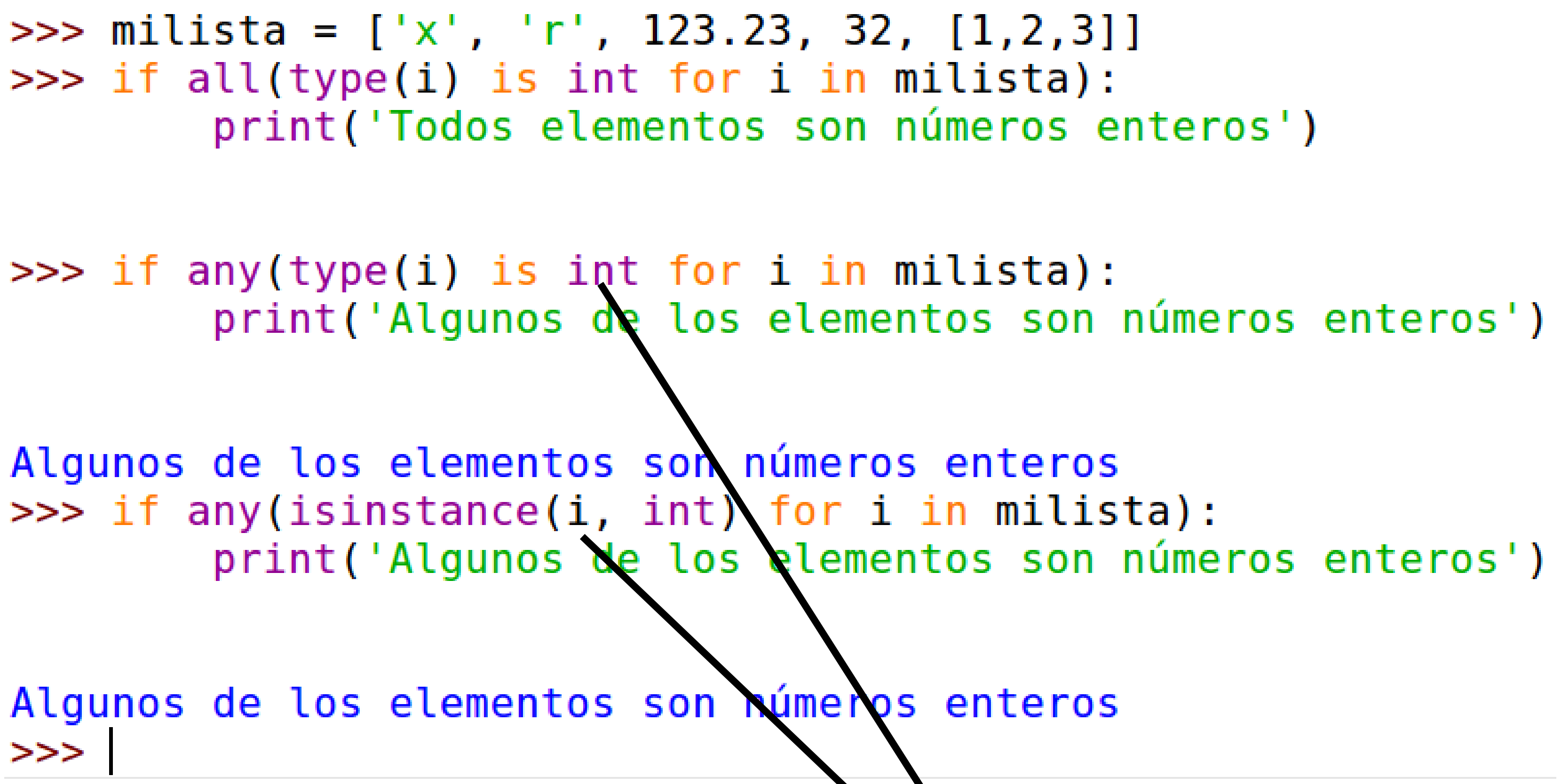
«

Ejemplos con all() y any()

```
>>> multiplos_de_6 = [i%6 == 0 for i in range(10)]
>>> multiplos_de_6
[True, False, False, False, False, False, True, False, False, False]
>>> all(multiplos_de_6)
False
>>> any(multiplos_de_6)
True
```

```
>>> numeros = [1, 2, -1, 9, -5]
>>> numeros_positivos = [x > 0 for x in numeros]
>>> numeros_positivos
[True, True, False, True, False]
>>> if all(numeros_positivos):
        print('Todos los números son positivos')


>>> if any(numeros_positivos):
        print('Algunos de los números son positivos')


Algunos de los números son positivos
```

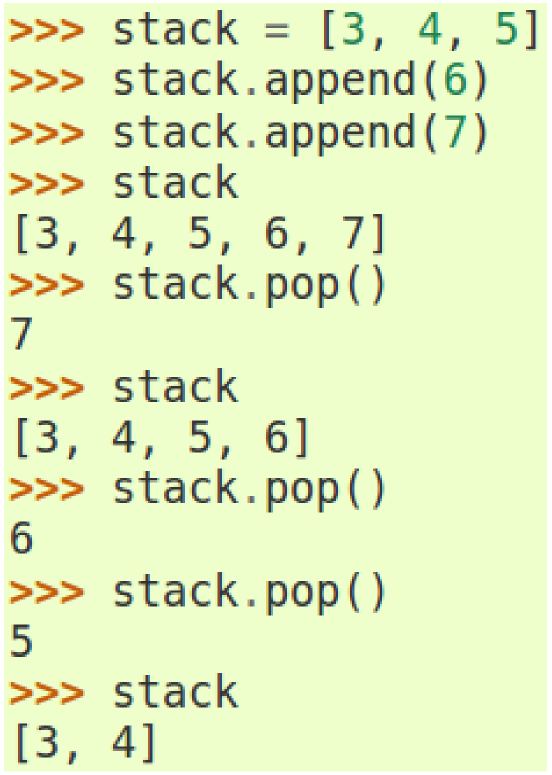
Ejemplos con all() y any()



```
>>> milista = ['x', 'r', 123.23, 32, [1,2,3]]
>>> if all(type(i) is int for i in milista):
        print('Todos elementos son números enteros')


>>> if any(type(i) is int for i in milista):
        print('Algunos de los elementos son números enteros')


Algunos de los elementos son números enteros
>>> if any(isinstance(i, int) for i in milista):
        print('Algunos de los elementos son números enteros')


Algunos de los elementos son números enteros
>>> 
```

Observe estas dos posibles formas de verificar los tipos de datos de los elementos

Usando una lista como una pila



```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

Usando una lista como una cola

It is also possible to use a list as a queue, where the first element added is the first element retrieved ("first-in, first-out"); however, lists are not efficient for this purpose. While appends and pops from the end of list are fast, doing inserts or pops from the beginning of a list is slow (because all of the other elements have to be shifted by one).

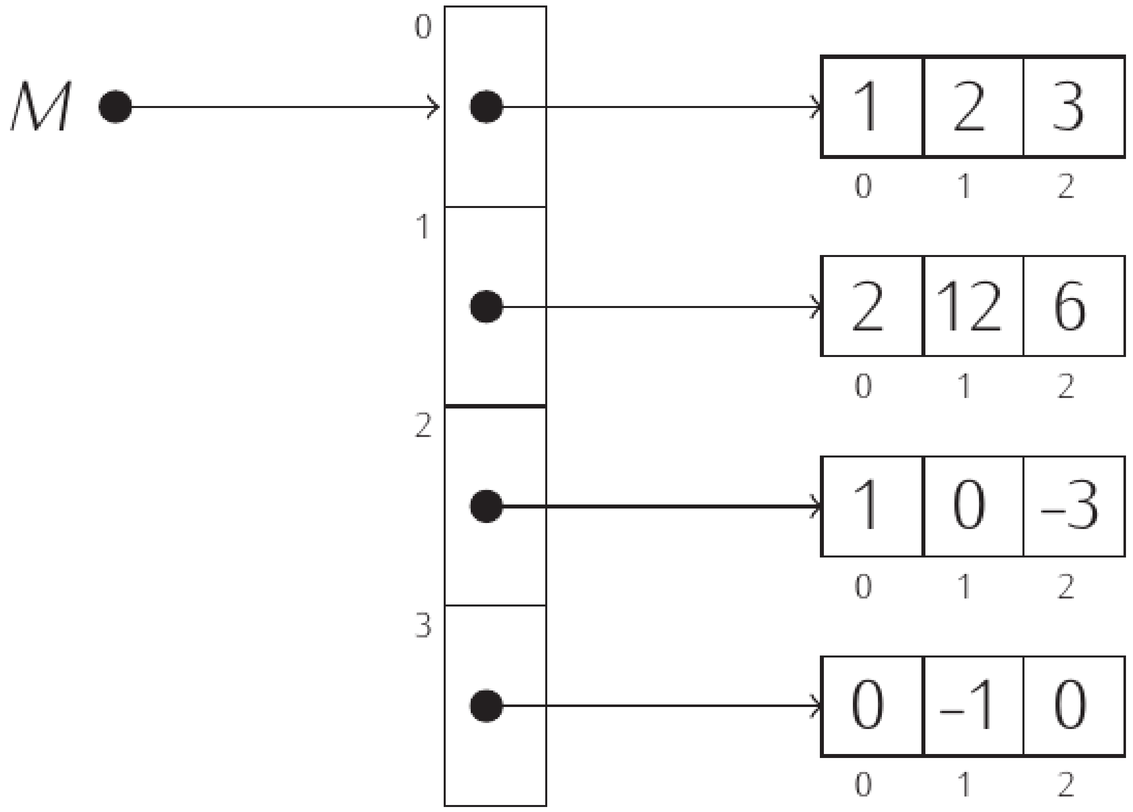
To implement a queue, use `collections.deque` which was designed to have fast appends and pops from both ends. For example:

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")          # Graham arrives
>>> queue.popleft()                 # The first to arrive now leaves
'Eric'
>>> queue.popleft()                 # The second to arrive now leaves
'John'
>>> queue                           # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

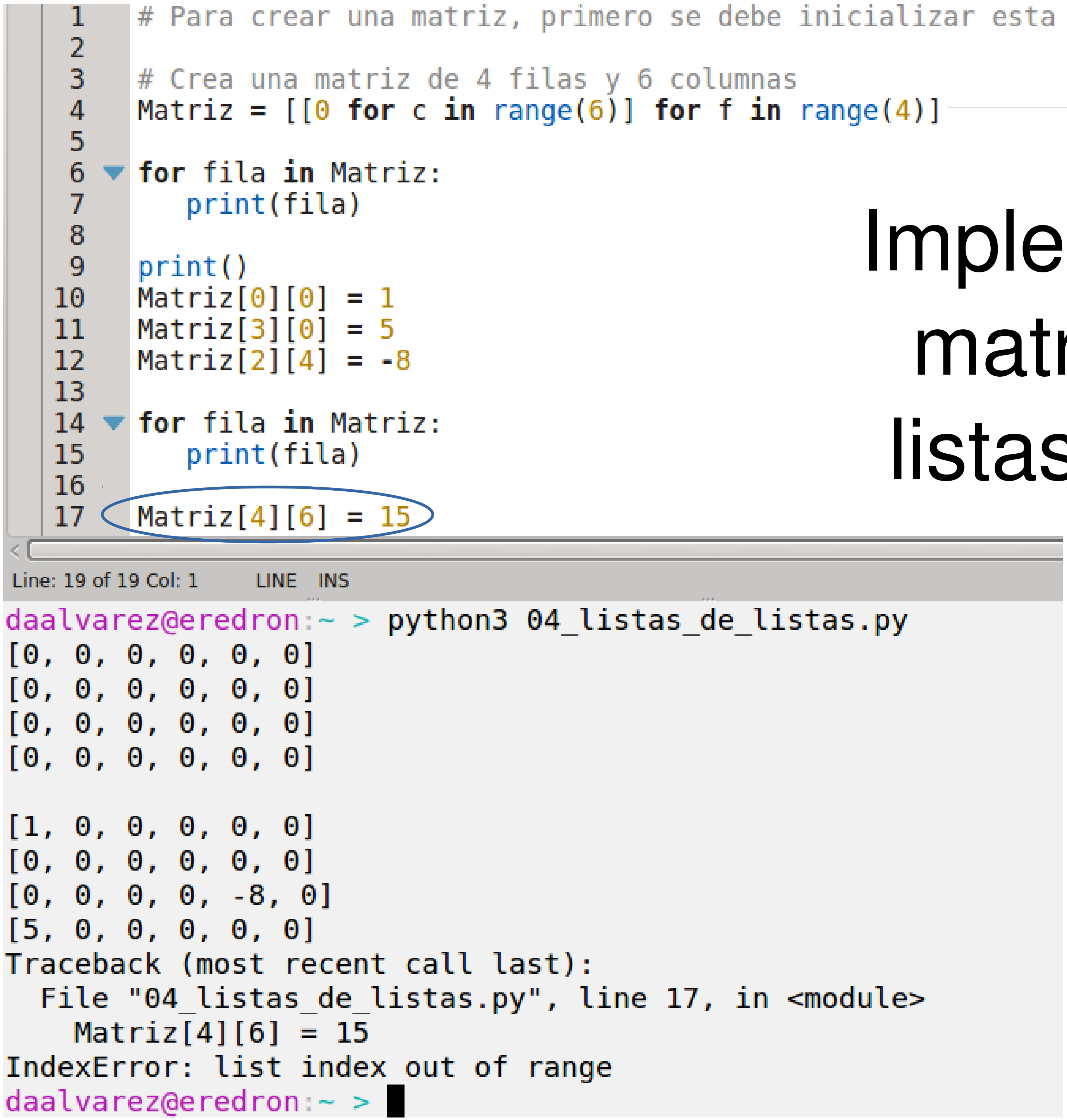
Matrices: listas de listas

```
M = [[1,  2,  3],
     [2, 12,  6],
     [1,  0, -3],
     [0, -1,  0]]
>>> M
[[1, 2, 3], [2, 12, 6], [1, 0, -3], [0, -1, 0]]
>>> |
```

$$M = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 12 & 6 \\ 1 & 0 & -3 \\ 0 & -1 & 0 \end{pmatrix}$$



Implementando matrices con listas de listas

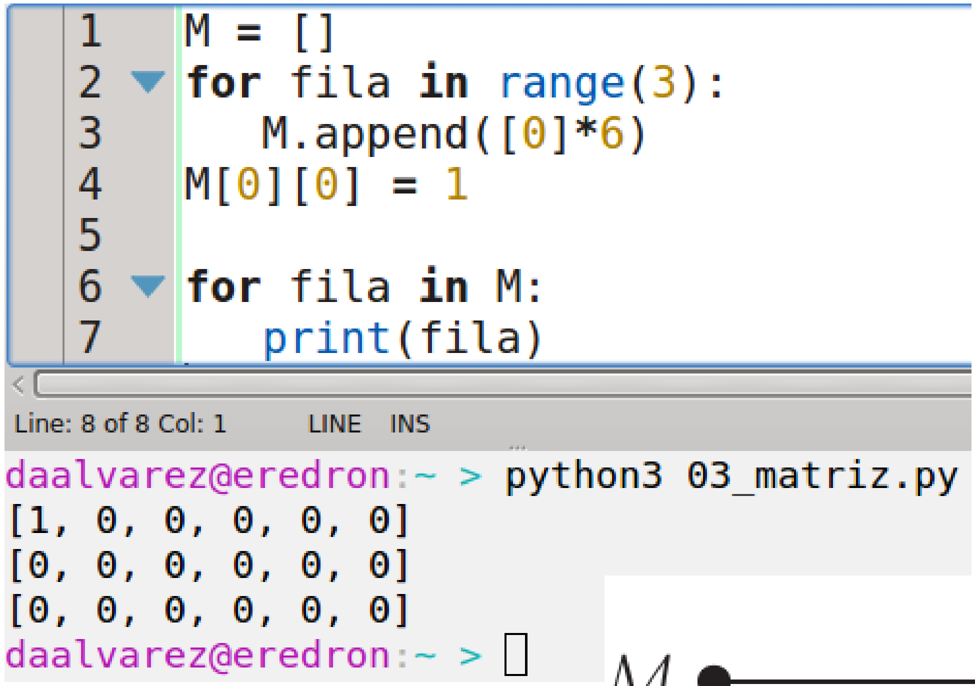


```
 1  # Para crear una matriz, primero se debe inicializar esta
 2
 3  # Crea una matriz de 4 filas y 6 columnas
 4  Matriz = [[0 for c in range(6)] for f in range(4)]
 5
 6  for fila in Matriz:
 7      print(fila)
 8
 9  print()
10  Matriz[0][0] = 1
11  Matriz[3][0] = 5
12  Matriz[2][4] = -8
13
14  for fila in Matriz:
15      print(fila)
16
17  Matriz[4][6] = 15
```

“nested list comprenhension”

```
daalvarez@eredron:~ > python3 04_listas_de_listas.py
[0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0]

[1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, -8, 0]
[5, 0, 0, 0, 0, 0]
Traceback (most recent call last):
  File "04_listas_de_listas.py", line 17, in <module>
    Matriz[4][6] = 15
IndexError: list index out of range
daalvarez@eredron:~ >
```

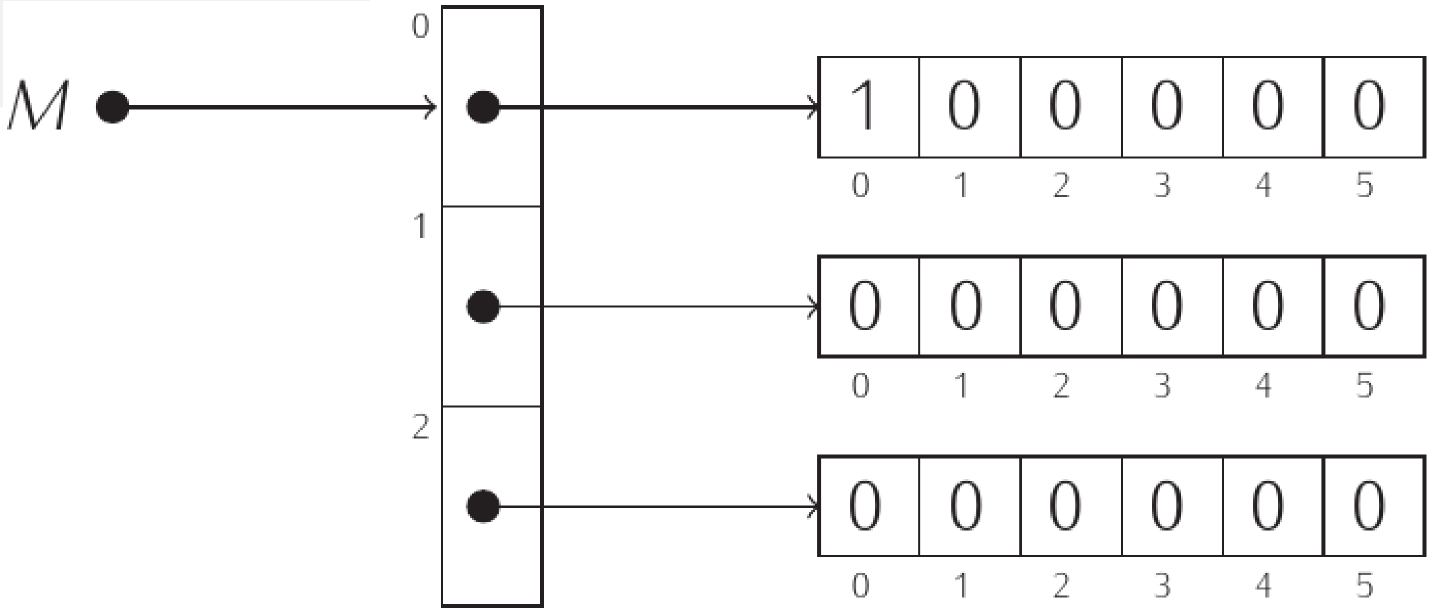
Otra forma correcta de crear las matrices



```python
M = []
for fila in range(3):
    M.append([0]*6)
M[0][0] = 1

for fila in M:
    print(fila)
```

Line: 8 of 8 Col: 1 LINE INS

```
daalvarez@eredron:~ > python3 03_matriz.py
[1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0]
daalvarez@eredron:~ >
```

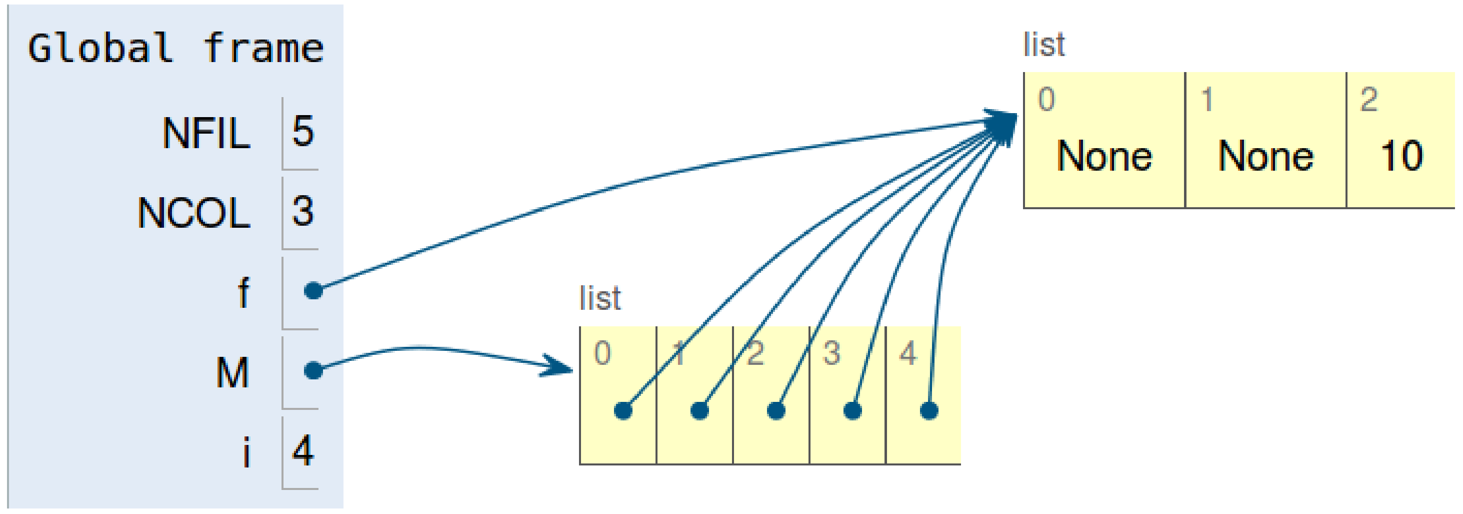


Forma incorrecta de crear matrices

```
NFIL = 5
NCOL = 3

f = NCOL*[None]
M = NFIL*[f]

M[1][2] = 10

for i in range(NFIL):
    print(M[i])
```

Program output:

```
[None, None, 10]
[None, None, 10]
[None, None, 10]
[None, None, 10]
[None, None, 10]
```

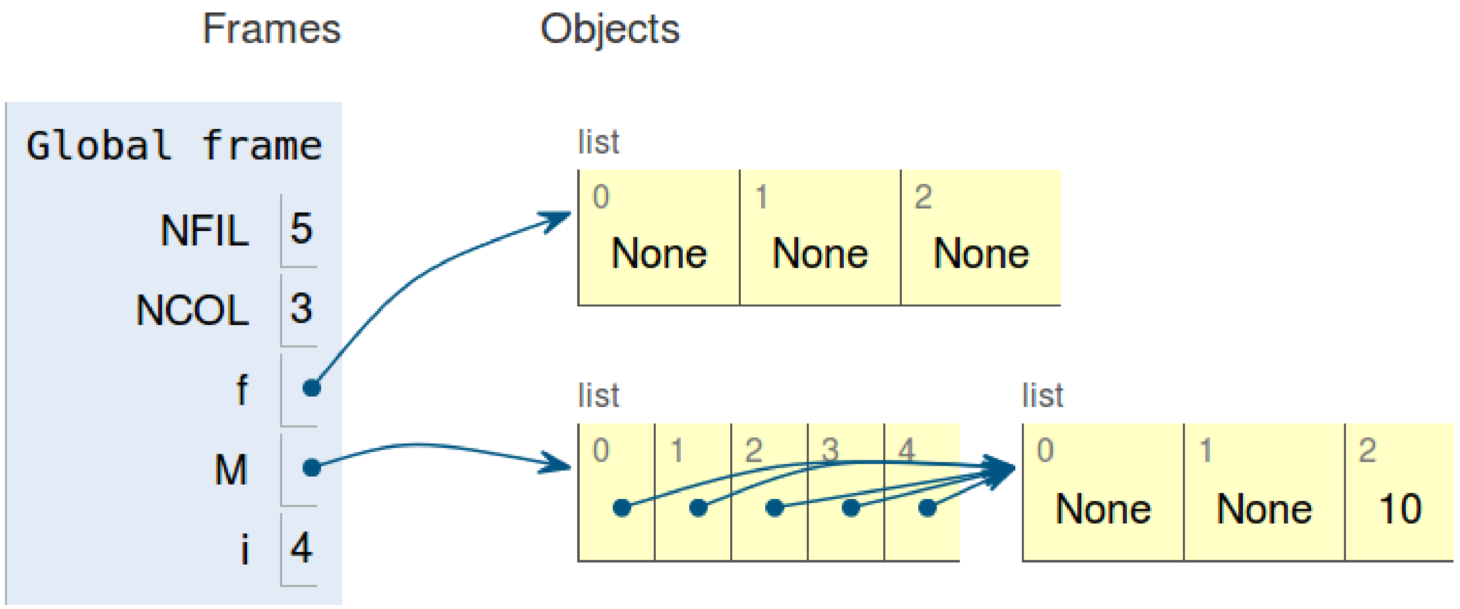


Otra forma incorrecta de crear matrices

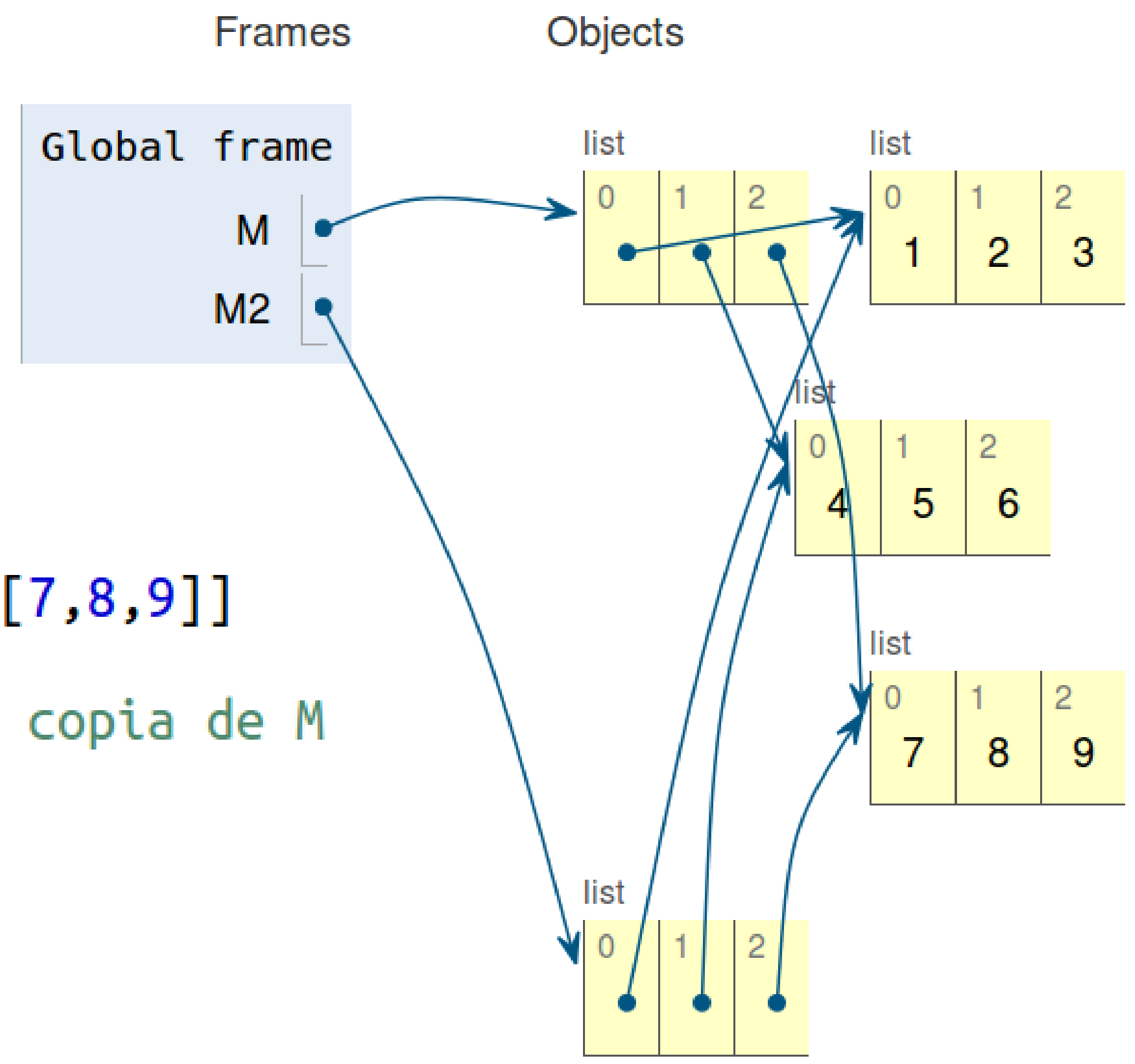
```python
NFIL = 5
NCOL = 3

f = NCOL*[None]
M = NFIL*[f[:]]

M[1][2] = 10

for i in range(NFIL):
    print(M[i])
```

Program output:

```
[None, None, 10]
[None, None, 10]
[None, None, 10]
[None, None, 10]
[None, None, 10]
```

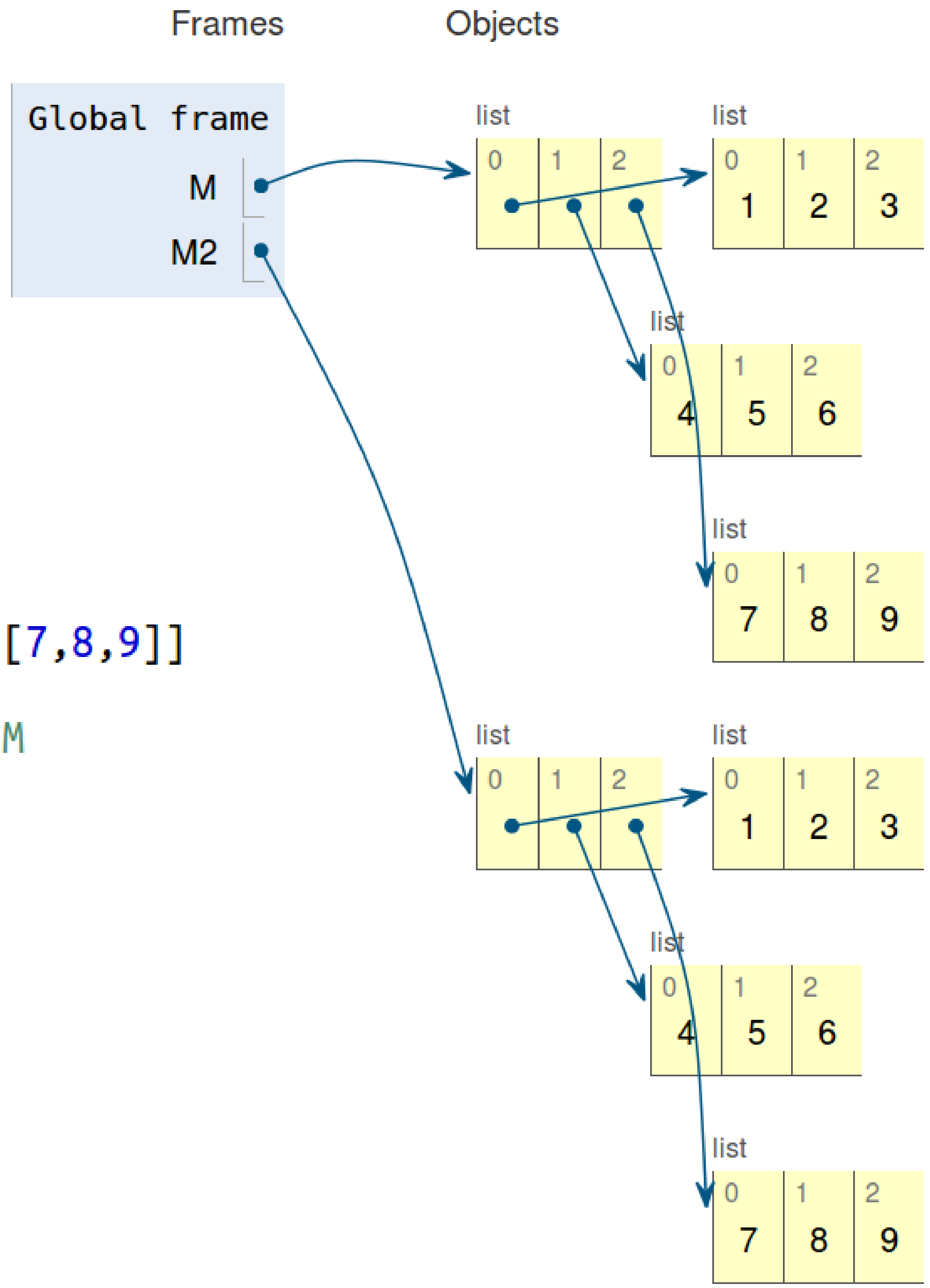


Forma incorrecta de copiar matrices



```python
M = [[1,2,3], [4,5,6], [7,8,9]]

# se pretende hacer una copia de M
M2 = M[:]
```

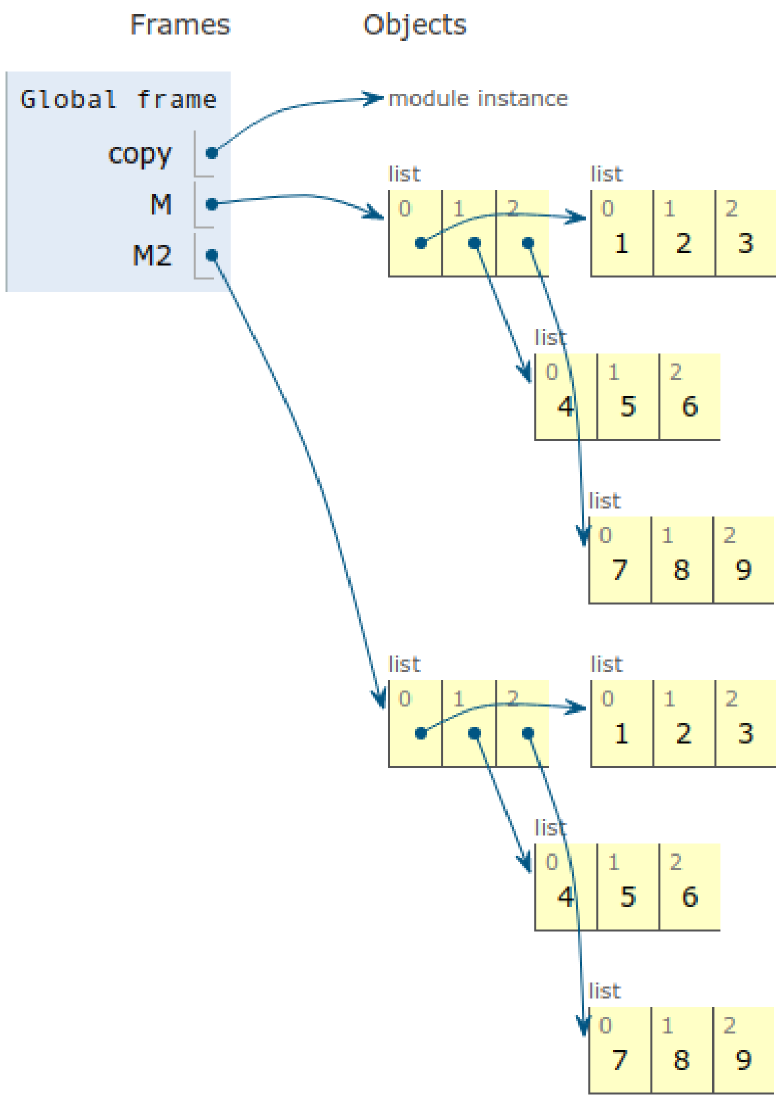
Forma correcta de copiar matrices



```
M = [[1,2,3], [4,5,6], [7,8,9]]

# se hace una copia de M
M2 = [f[:] for f in M]
```

Copiando matrices usando “deep copy”

```python
import copy

M = [[1,2,3], [4,5,6], [7,8,9]]

M2 = copy.deepcopy(M)
```



Shallow copy vs Deep copy

The difference between shallow and deep copying is only relevant for compound objects (objects that contain other objects, like lists or class instances):

- A **shallow copy** constructs a new compound object and then (to the extent possible) inserts references into it to the objects found in the original. Shallow copies duplicate as little as possible. A shallow copy of a collection is a copy of the collection structure, not the elements. With a shallow copy, two collections now share the individual elements. Se hace usando el método copy(), el operador de slicing [:] o con copy.copy().
- A **deep copy** constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original. Deep copies duplicate everything. A deep copy of a collection is two collections with all of the elements in the original collection duplicated. Se hace con copy.deepcopy()

Tamaño de las variables en memoria

- Varían de implementación a implementación del interpretador.

```
>>> import sys
>>> x = 123
>>> sys.getsizeof(x)
28
>>> sys.getsizeof(123)
28
>>> sys.getsizeof('abc')
52
>>> sys.getsizeof('abcd')
53
>>> x = 2**1000
>>> sys.getsizeof(x)
160
>>> sys.getsizeof([1, 2, 3, 'r'])
96
```

Notas finales

- Las listas de python son análogas a la funcionalidad que provee el comando cell() de MATLAB
- Cuando se tienen listas de datos en las que todas las entradas son del mismo tipo, es frecuentemente mucho más eficiente usar la librería **numpy** y su tipo de dato **array**. Un array de numpy es un bloque de memoria contigua en la cual se almacenan datos del mismo tipo; esto permite hacer operaciones sobre los elementos de forma más veloz ya que los elementos se tienen un acceso mucho más fácil en memoria, y python los accesa de esta forma mucho más rápido. Adicionalmente, la mayoría de las rutinas de numpy están escritas en lenguaje C.

Referencias

- Wikipedia
- http://www.inventwithpython.com/
- http://www.diveintopython3.net/
- Documentación de Python:
  - https://docs.python.org/3/tutorial/index.html
  - https://docs.python.org/3/
- Marzal Varó, Andrés; Gracia Luengo, Isabel; García Sevilla, Pedro (2014). Introducción a la programación con Python 3. Disponible en: http://dx.doi.org/10.6035/Sapientia93
- https://realpython.com/python-lists-tuples/