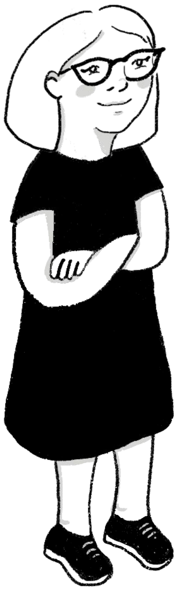
List me your virtues

Mai Giménez

PYCONES 2018 MÁLAGA

PYLONDINIUM 2018

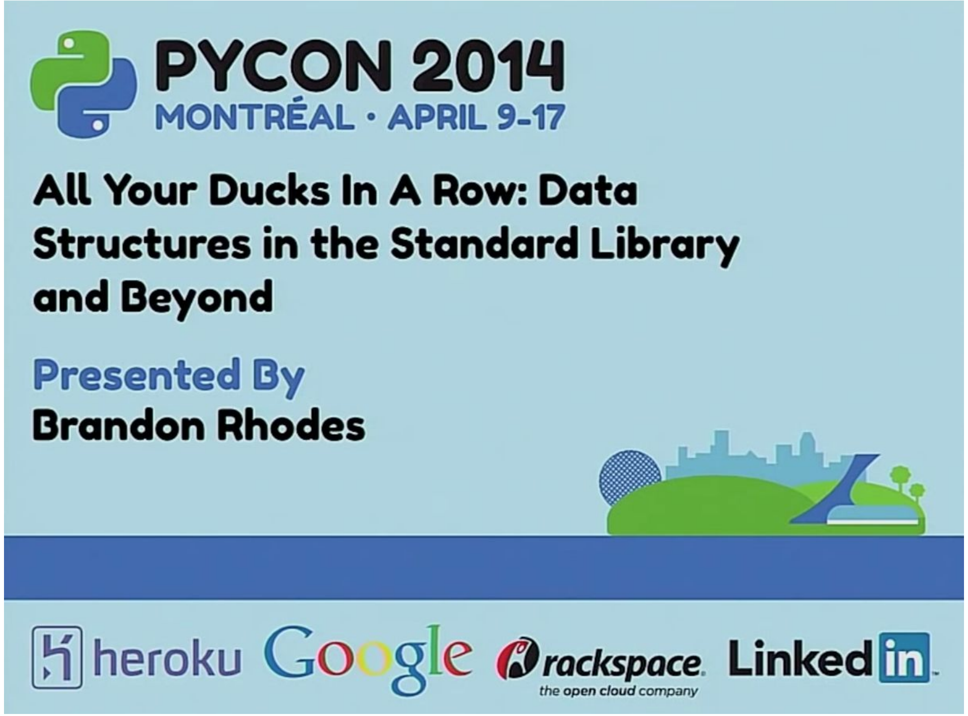


OH, HELLO! I'M MAI & WE ARE GOING TO HAVE FUN LEARNING

Mai Giménez
hi@maigimenez.es
maidotgimenez@twitter
maigimenez@github


HAPPY
PROUD
JUNE

RHODES KNOWS IT BETTER



All your ducks in a row
Brandon Rhodes
PyCon 2014

“THE LIST IS
THE MOST DANGEROUS
DATA STRUCTURE”

Brandon Rhodes


WHAT?!
#DragRace
Season Premiere
VH1

1.

THE LIST CONSTRUCTOR

LISTS 101

_ A list is a data structure that contains a sequence of elements of the same type ordered.
_ Lists are a mutable object.
_ The constructor of a list:

```
list([iterable_object])
```

```
[]
```

OK, GOT IT. LET'S CREATE A LIST.

_ Let's create some lists then:

```
>>> list("hello world")
```

h	e	l	l	o		w	o	r	l	d														

```
>>> [1,2,3]
```

1	2	3					

THERE SHOULD BE ONE
- AND PREFERABLY ONLY ONE -
OBVIOUS WAY TO DO IT.

Python zen

LIST() VS []: THE PYTHONIC WAY

_ The Pythonic (and fastest) way to do it is using the **[]**

```
>>> timeit("[]")
  0.04296872499980964

>>> timeit("list()")
  0.20007910901040304
```

LIST() VS []: LITERALS

_ List is a built-in object whereas [ ] is a literal* . Hence, a list has to look up for its namespace but not the [ ]

```
>>> dis.dis(lambda : [])
    1           0 BUILD_LIST               0
                2 RETURN_VALUE

>>> dis.dis(lambda : list())
    1           0 LOAD_GLOBAL              0 (list)
                2 CALL_FUNCTION            0
                4 RETURN_VALUE
```

* A ***literal*** is a succinct form of writing a built-in object that the parser can recognise easily.

LIST() VS []: THE NOT SO OBVIOUS BEHAVIOURS

_ Sometimes the behaviour might not be what we expected

```
>>> list(“abc”)
    [“a”, “b”, “c”]

>>> [“abc”]
    [“abc”]
```

_ From the language definition: **list_display** ::= “[” [starred_list | comprehension]“]”

_ Also, from the Python documentation: **class list([iterable]):** The constructor builds a list whose items are the same and in the same order as *iterable*’s items.

LIST() VS []: THE NOT SO OBVIOUS BEHAVIOURS

_ If *iterable* is already a list, a copy is made and returned using the constructor, but not using the literal.

```
>>> list_1 = [1,2,3]
    [1,2,3]
>>> id(list_1)
    4479653128

>>> list_2 = list(list_1)
    [1,2,3]
>>> id(list_2)
    4473062408
```

```
>>> list_1 = [1,2,3]
    [1,2,3]
>>> id(list_1)
    4479653128

>>> list_2 = [list_1]
    [[1,2,3]]
>>> id(list_2)
    447315328
>>> id(list_2[0])
    4479653128
```

LIST() VS []: THE NOT SO OBVIOUS BEHAVIOURS

_ You could overwrite the built-in

```
>>> import builtins
>>> builtins.list = set

>>> list()
    set([])

>>> []
    []
```

_ Shout out to @pyblogsal

2.

INDEXING LISTS

INDEXING LISTS

_ An item in a list can be accessed using its index, an integer value that tags each position starting from 0. But we could also access counting backwards.

```
>>> hello_list = list(“hello world”)
```

0	1	2	3	4	5	6	7	8	9	10
h	e	l	l	o		w	o	r	l	d
-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

```
>>> hello_list[4]
    o
```

```
>>> hello_list[-3]
    r
```

INDEXING LISTS

_ The cost of accessing an item is constant **O(1)**.

_ If we have an array where all its elements are the same, it's easy to explain how this can be done.

hello_list => 0x00

characters => 1B *

i-th character => 0X00 + (i * 8b)

```
>>> hello_list[4]
```

_ But lists in Python can have multiple objects, How did Python know where to find the element in O(1) time?

* Let's roll with this unicode oversimplification

3.

THE CPYTHON LIST

THE CPYTHON LIST ON BARE METAL

_ If we create an empty list is not completely empty.

```
>>> len([])
    0

>>> import sys
>>> sys.getsizeof([])
    64
```

HOW LISTS ARE IMPLEMENTED

What the documentation says:

_ Python's lists are really variable-length arrays, *not Lisp-style linked lists*.

_ The implementation uses a **contiguous array of references to other objects** and keeps a pointer to this array and the array's length in a list head structure.

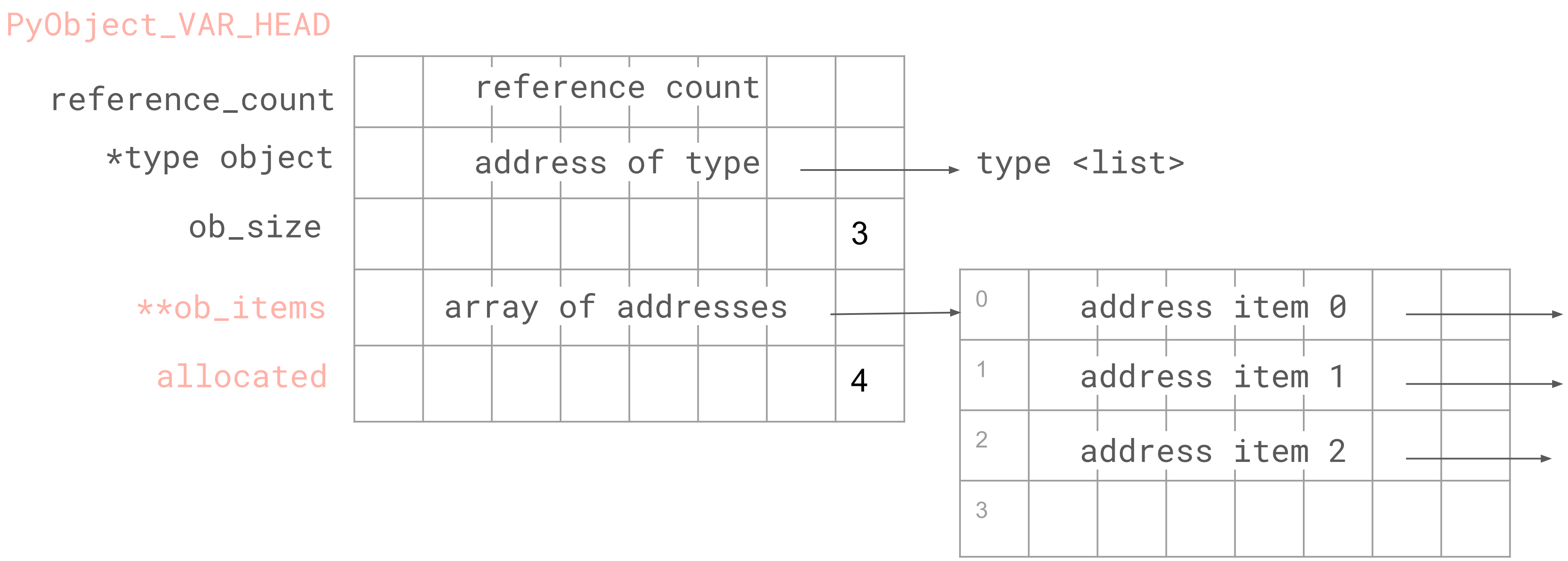
_ When items are appended or inserted, the array of references is resized. *Some cleverness is applied* to improve the performance of appending items repeatedly; when the array must be grown, some extra space is allocated so the next few times don't require an actual resize.

THE CPYTHON LIST ON BARE METAL

What the code says:

```c
typedef struct {
   // Macro used when declaring new types with a varying length.
   PyObject_VAR_HEAD;
   // Vectors of pointers to a list of items.
   PyObject **ob_item;
   // Memory allocated.
   Py_ssize_t allocated;
} PyListObject;
```

THE CPYTHON LIST ON BARE METAL



TEST ME!

```
>>> init_list = [1]

>>> append_list = []
>>> append_list.append(1)

>>> assert sys.getsizeof(list_init) == sys.getsizeof(list_append)
```

TEST ME!

```
>>> init_list = [1]

>>> append_list = []
>>> append_list.append(1)

>>> assert sys.getsizeof(list_init) == sys.getsizeof(list_append)
    ------------------------------------------------------------
    AssertionError
```

TEST ME!

```
>>> init_list = [1]

>>> append_list = []
>>> append_list.append(1)

>>> assert sys.getsizeof(list_init) == sys.getsizeof(list_append)
    --------------------------------------------------------------
    AssertionError


>>> assert sys.getsizeof(list_init)
    72

>>> assert sys.getsizeof(append_list)
    96
```


VH1

INIT VS APPEND

_ **Empty list**: we need 64 B for storing the list data structure

```
>>> sys.getsize([])
    64
```

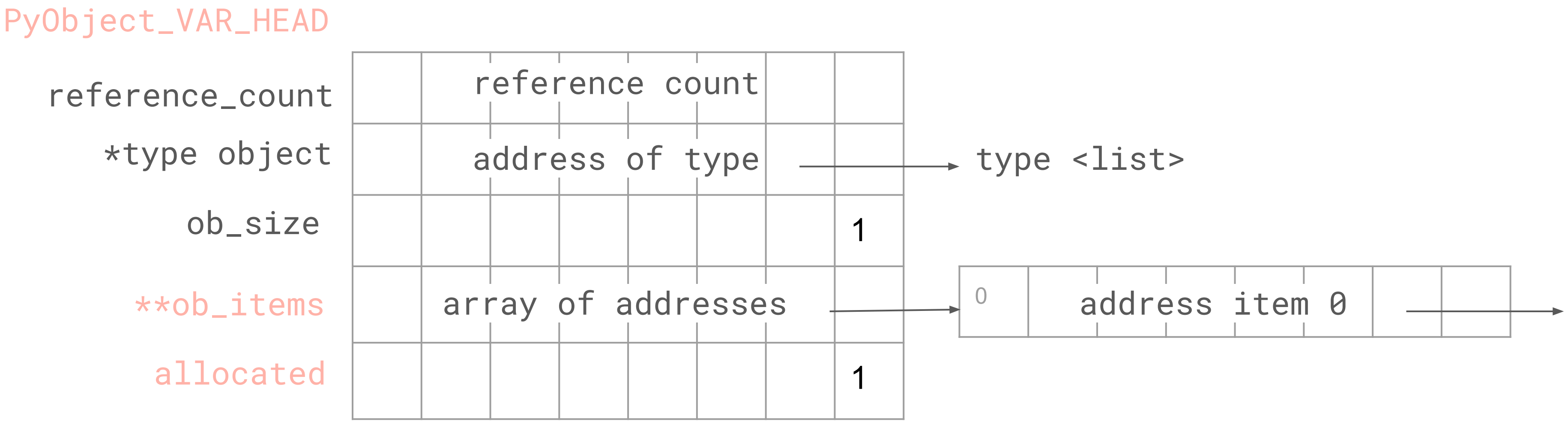
_ **List with one element**: in this case we are going to store space for the element as well, 8B for an pointer in a 64b machine.

```
>>> sys.getsize([1])
    72
```

* Eli Bendersky explains it deeper in this Stackoverflow answer.

INIT

```
>>> init_list = [1]
```



APPEND

_ **Append an element to a list**: if we want to append an element to an empty list we will need to allocate new memory and here is where the cleverness happens. The amortized cost is O(1)

```
function list_resize:
     input: list_object, new_size
     output: 0 if OK -1 otherwise

     if (allocated >= new_size && new_size >= (allocated << 1))
          // Do not reallocate memory
          return 0;

     // The growth pattern is: 0, 4, 8, 16, 25, 35, 46, ...
     new_allocated = new_size + new_size >> 3 + (new_size < 9 ? 3 :6);

     // Do not overflow
     if (new_allocated > PY_SIZE_MAX - new_size)
          PyError_NoMemory(); return -1

     reallocate_each_item()
```

APPEND

_ **Append an element to a list**: if we want to append an element to an empty list we will need to allocate new memory and here is where the cleverness happens. The amortized cost is O(1)

Cleverness! Bypass realloc

```
function list_resize:
     input: list_object, new_size
     output: 0 if OK -1 otherwise

     if (allocated >= new_size && new_size >= (allocated << 1))
          // Do not reallocate memory
          return 0;

     // The growth pattern is: 0, 4, 8, 16, 25, 35, 46, ...
     new_allocated = new_size + new_size >> 3 + (new_size < 9 ? 3 :6);

     // Do not overflow
     if (new_allocated > PY_SIZE_MAX - new_size)
          PyError_NoMemory(); return -1

     reallocate_each_item()
```

INIT VS APPEND

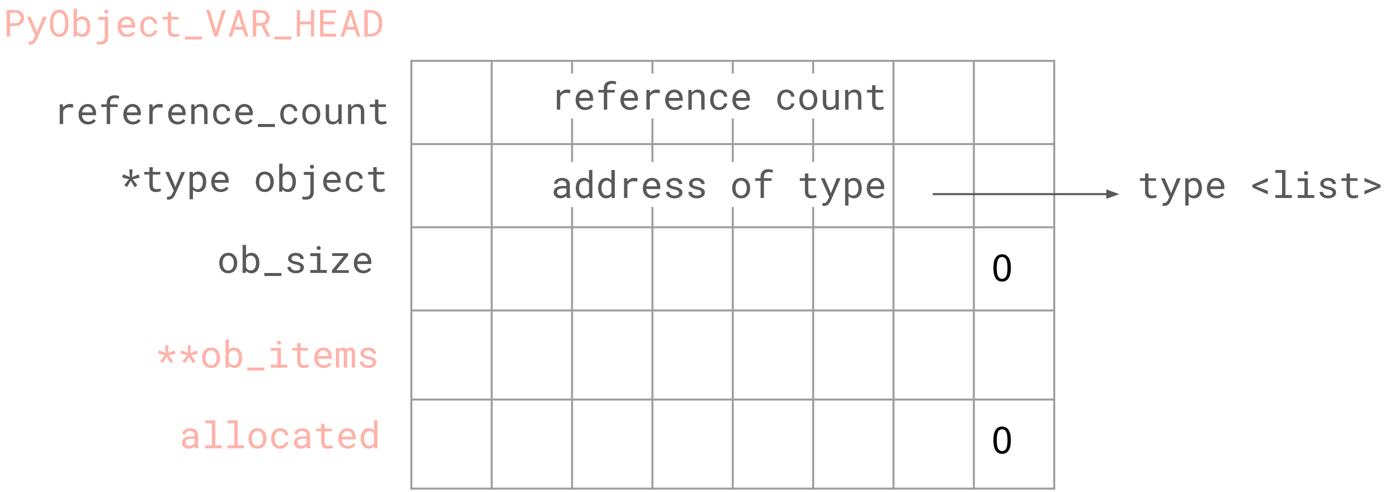
_ **Append an element to a list**: if we want to append an element to an empty list we will need to allocate new memory and here is where the cleverness happens. The amortized cost is O(1)

```
function list_resize:
     input: list_object, new_size
     output: 0 if OK -1 otherwise

     if (allocated >= new_size && new_size >= (allocated << 1))
          // Do not reallocate memory
          return 0

     // The growth pattern is: 0, 4, 8, 16, 25, 35, 46, ...
     new_allocated = new_size + new_size >> 3 + (new_size < 9 ? 3 : 6)

     // Do not overflow
     if (new_allocated > PY_SIZE_MAX - new_size)
          PyError_NoMemory(); return -1

     reallocate_each_item()
```

Cleverness! Mild reallocation

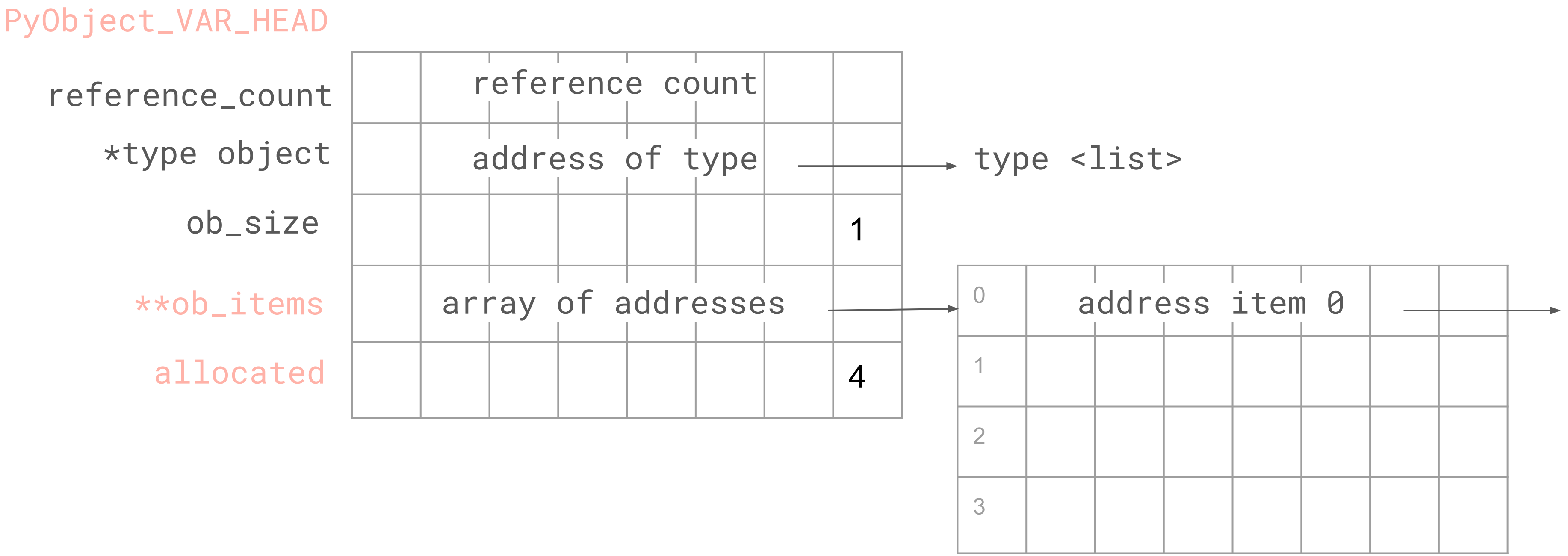
APPEND

```
>>> append_list = []
```



APPEND

```
>>> append_list.append(1)
```



4.

LIST OF DANGERS


NO ONE IS SAFE

THE ONE-LINER OFFENDERS

_Some operations seem to be single operations—hence O(1)— that are actually operating over multiple items.

`l[i]`	O(1)		
`l[i] = v`	O(1)		
		`l.extend(k)`	O(k)
`append(v)`	O(1)	`append(j)`	O(n)
`insert(len(l), i)`	O(1)	`insert(0, i)`	O(n)
`del l[-1]`	O(1)	`del l[0]`	O(n)
`l.pop()`	O(1)	`l.pop(0)`	O(n)
`v in l`	O(1)	`v in l`	O(n)

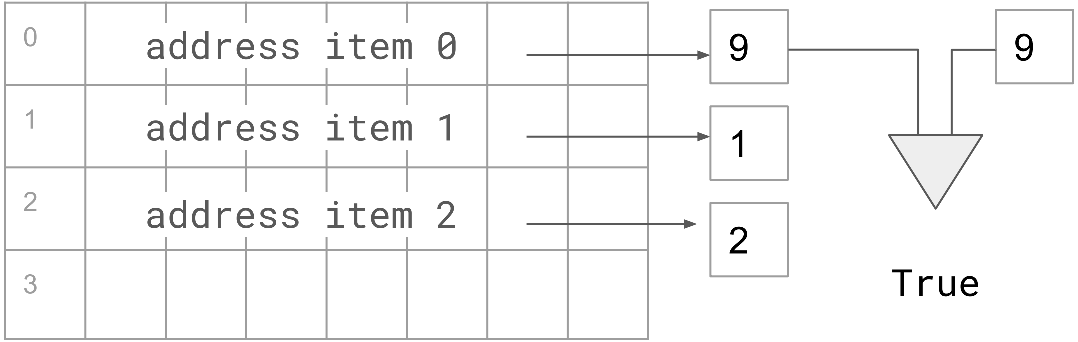
4.1.

IN

DANGEROUS OPERATIONS: IN

_ In order to make sure that the element is in the list we will need to check all the elements.
Hence, the average cost is O(n)

```
>>> 9 in [9,2,3]
```

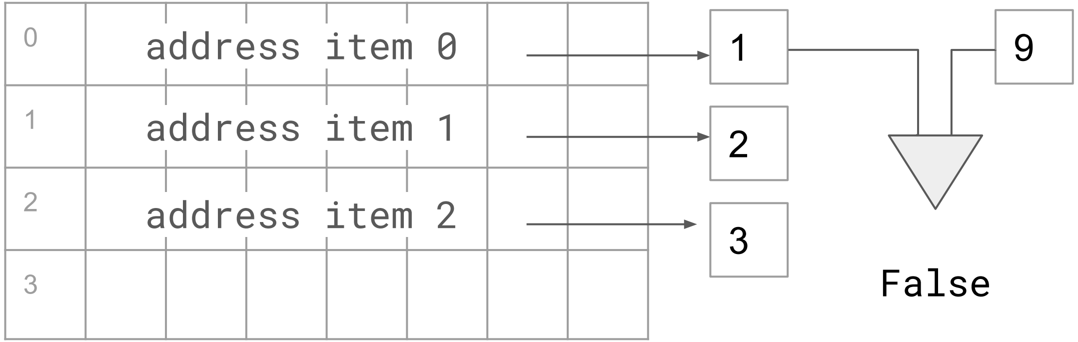



Party

DANGEROUS OPERATIONS: IN

_ In order to make sure that the element is in the list we will need to check all the elements. Hence, the average cost is O(n)

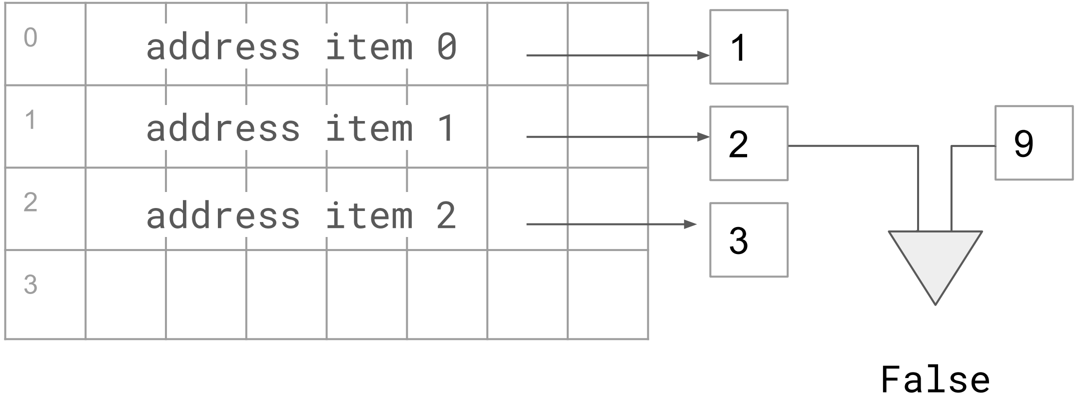
```
>>> 9 in [1,2,3]
```



DANGEROUS OPERATIONS: IN

_ In order to make sure that the element is in the list we will need to check all the elements. Hence, the average cost is O(n)

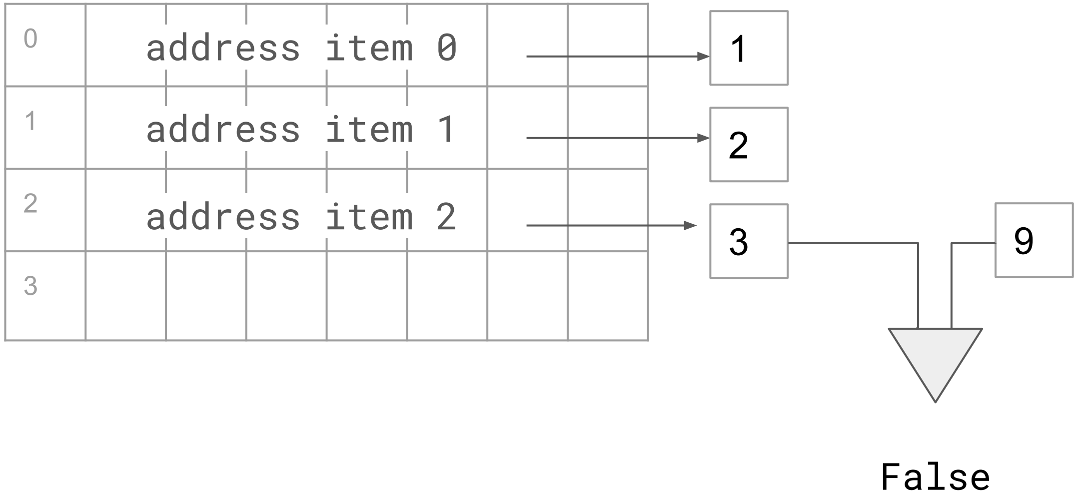
```
>>> 9 in [1,2,3]
```



DANGEROUS OPERATIONS: IN

_ In order to make sure that the element is in the list we will need to check all the elements. Hence, the average cost is O(n)

```
>>> 9 in [1,2,3]
```



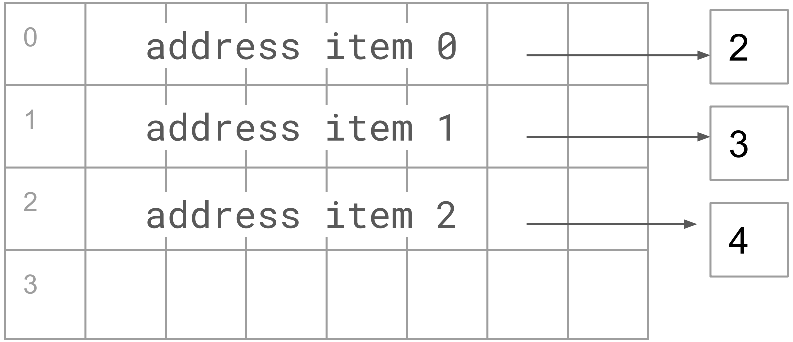
4.2.

INSERT

DANGEROUS OPERATIONS: INSERT

_ If we don´t insert at the end of the list, the cost of inserting an item v in position p is O(n)

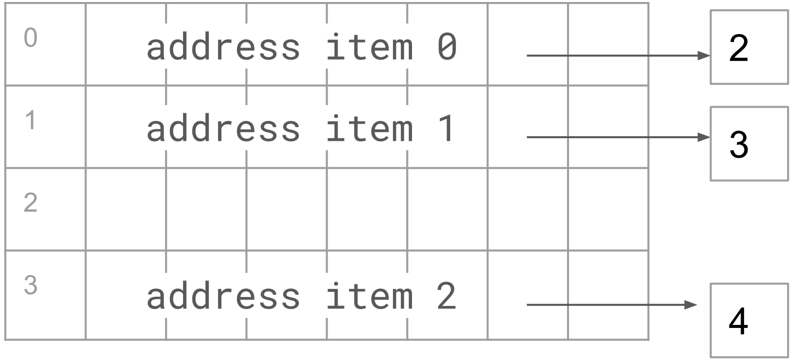
```
>>> ins_list = [2,3,4]
>>> ins_list.insert(1,0)
```



DANGEROUS OPERATIONS: INSERT

_ If we don´t insert at the end of the list, the cost of inserting an item v in position p is O(n)

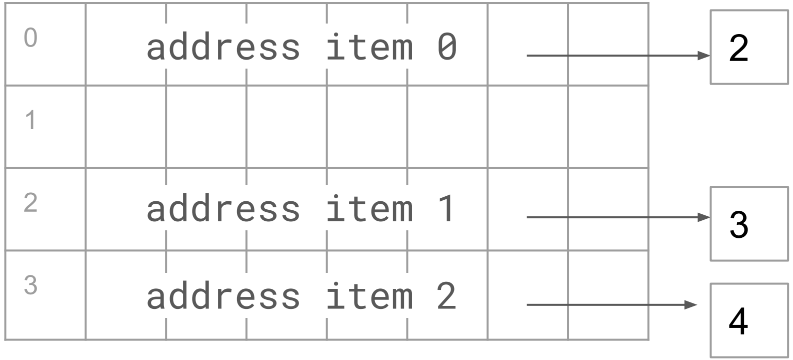
```
>>> ins_list = [2,3,4]
>>> ins_list.insert(1,0)
```



DANGEROUS OPERATIONS: INSERT

_ If we don´t insert at the end of the list, the cost of inserting an item v in position p is O(n)

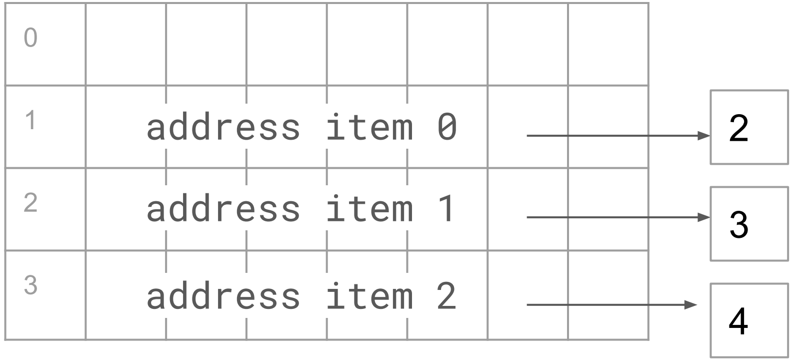
```
>>> ins_list = [2,3,4]
>>> ins_list.insert(1,0)
```



DANGEROUS OPERATIONS: INSERT

_ If we don´t insert at the end of the list, the cost of inserting an item v in position p is O(n)

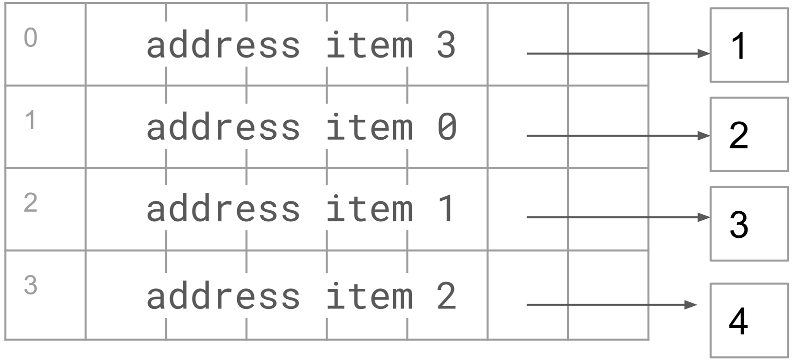
```
>>> ins_list = [2,3,4]
>>> ins_list.insert(1,0)
```



DANGEROUS OPERATIONS: INSERT

_ If we don´t insert at the end of the list, the cost of inserting an item v in position p is O(n)

```
>>> ins_list = [1,3,4]
>>> ins_list.insert(2,0)
```



4.3.

THE SLICING OPERATOR :

THE TERRIFYING SLICING OPERATOR

_ Slicing is really tricky because it selects a range of items in a sequence and creates a new list, copying all the elements from the slice.

```
>>> sliceable_list = [1, 2, 3, 4, 5, 6]
>>> part_list = sliceable_list[2:4]
>>> part_list
    [3, 4]
```

_ Notation

```
slice := slicing “[“ [lower_bound]:[upper_bound] [“:”[stride]] “]”
```

THE TERRIFYING SLICING OPERATOR

_ **Slicing**: creates a new list and copy all the values in the segment from the source list to the new list.

```
function list_slice:
     input: src_list_object, low_offset, high_offset
     output: dst_list_object

     // Find the lower and the high offset if they're undefined.
     // Define the length and create the new list
     len = high_offset - low_offset
     np = new list(len)
     if np == NULL: return NULL

     // Copy items in the new array
     for i, item in enumerate(src_list_object):
          np[i] = item
          increment_reference(item)
```

THE TERRIFYING SLICING OPERATOR

_ Slicing allow us to get parts of a list easily.

```
>>> hello_list = list(“hello world”)

>>> hello_list[4:8]
    [“o”, “ ”, “w”, “o”]

>>> hello_list[6:]
# hello_list[6:len(hello_list),1]
    [“w”, “o”, “r”, “l”, “d”]

>>> hello_list[:-3]
# hello_list[0:-3,1]
    [“r”, “l”, “d”]
```

0	1	2	3	4	5	6	7	8	9	10
h	e	l	l	o		w	o	r	l	d
-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

THE TERRIFYING SLICING OPERATOR

_ Examples of slicing [start:end:stride]. If the stride is a negative value and we don't specify the start of the end, these will be the end (len(list)) and the beginning of the list (0) respectively::

```
# Reverse a list making a copy
>>> hello_list[::-1]
# hello_list[len(hello_list):0,-1]
    [“d”, ”l”, ”r”, ”o” , ”w”, ” ” “o”, ”l”, ”l”, ”e”, ”h”]

# Reverse a list making a copy inplace
>>> hello_list.reverse()

>>> hello_list[:2:-2]
# hello_list[len(hello_list):2,1]
    [“d”,”r”,”w”,”o”]
```

SLICING VS DEEPCOPY

_ Copying a list can be easily achieved through the usage of the slice operation.

```
>>> src_list = [1, 2, 3, 4, 5, 6]
>>> copy_list = src_list[:]
>>> print(copy_list)
    [1, 2, 3, 4, 5, 6]
>>> id(src_list)
    4446869896
>>> id(copy_list)
    4446829192
```

SLICING VS DEEPCOPY

_ But if the list contains another list, it will copy the reference not the list. Use deepcopy instead

```
>>> src_list = [[1, 2, 3], [4, 5, 6]]
>>> copy_list_ref = src_list[:]
>>> print(copy_list_ref)
    [[1, 2, 3], [4, 5, 6]]
>>> src_list[0][0] = 1
>>> print(copy_list_ref)
    [[0, 2, 3], [4, 5, 6]]

>>> import copy
>>> copy_list = copy.deepcopy(src_list)
```

4.3.

REMOVING FLAVOURS

DANGEROUS OPERATIONS: REMOVE

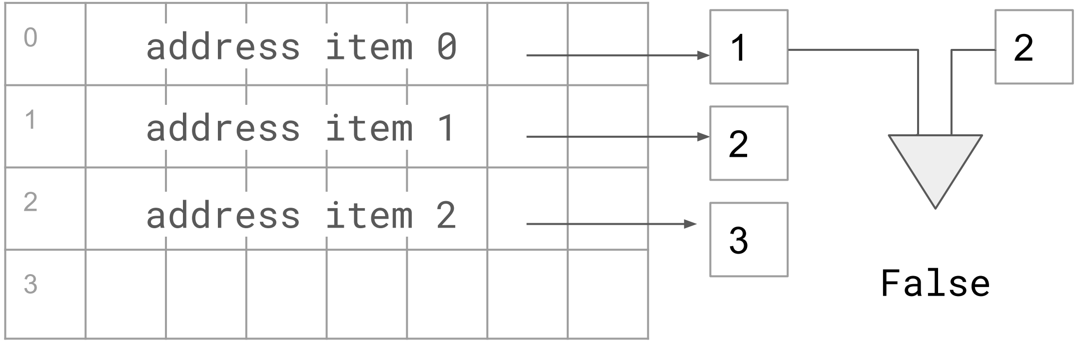
_ **Remove an element from a list**: removes the first matching element from a list. Slicing is called for removing the element.

```
function list_remove:
     input: list_object, value
     output: None if OK, NULL otherwise

     // Find the element to delete in the list
     for i in range(len(list_object):
          if comp(list_object[i], value) is True:

               // Slice the list to remove, recycle elements to delete
               // and resize the list
               if list_ass_slice(i, i+1):
                    return None

     //If the element is not in the list
     raise ValueError
     return NULL
```

DANGEROUS OPERATIONS: REMOVE

_ Removes an element from a list given its index

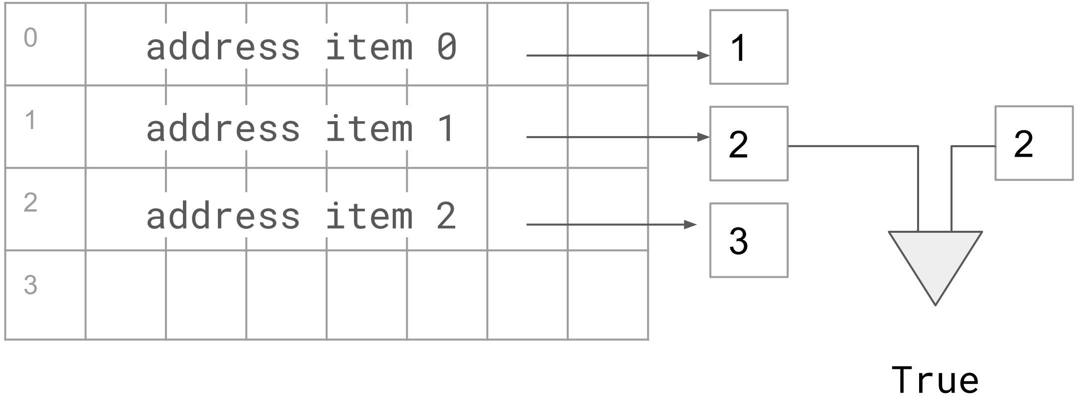
```
>>> rm_list = [1,2,3]
>>> remove.rm_list(1)
```



DANGEROUS OPERATIONS: REMOVE

_ Removes an element from a list given its index

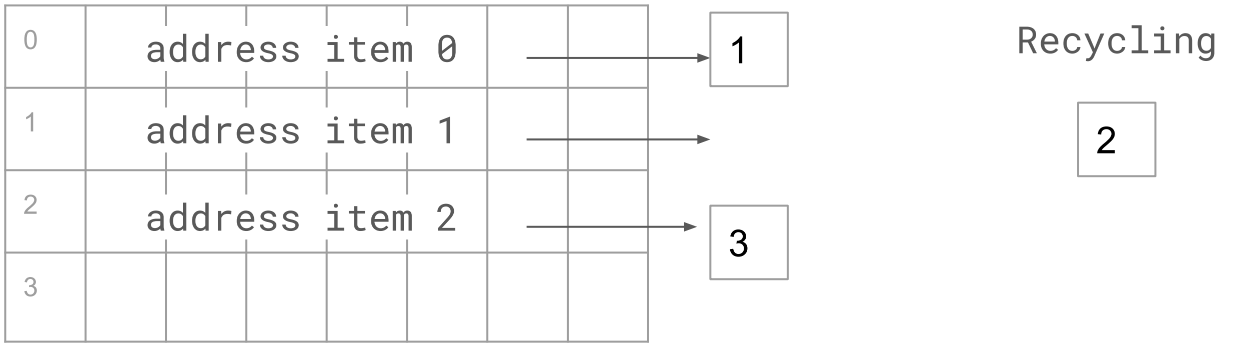
```
>>> rm_list = [1,2,3]
>>> remove.rm_list(1)
```



DANGEROUS OPERATIONS: REMOVE

_ Removes an element from a list given its index

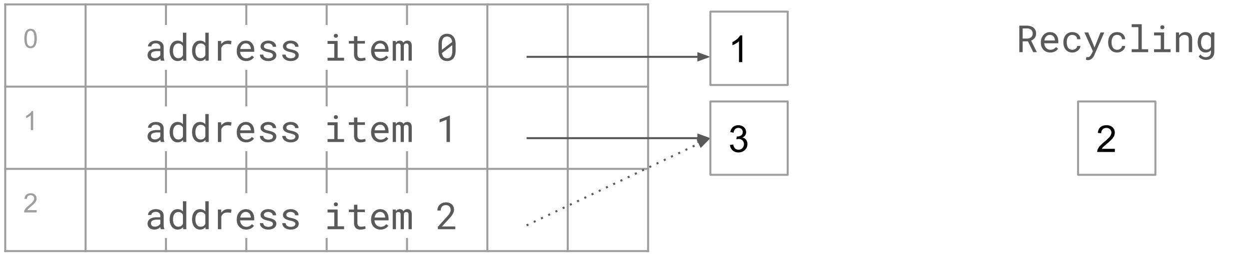
```
>>> rm_list = [1,2,3]
>>> remove.rm_list(1)
```



DANGEROUS OPERATIONS: REMOVE

_ Removes an element from a list given its index

```
>>> rm_list = [1,2,3]
>>> remove.rm_list(1)
```



THE DELETING CERBERUS: DELETE, REMOVE, POP.

_ There are differences between these operators but all of them are potentially dangerous, meaning O(n) .

_ **Remove**: removes the first matching item. O(n)

```
>>> rm_list = [1,2,3,2,4,2]
>>> remove.rm_list(2)
>>> print(rm_list)
    [1,3,2,4,2]
```

_ **del**: removes item in the position of the index given. O(n)

```
>>> del rm_list(3)
>>> print(rm_list)
    [1,2,3,4,2]
```

THE DELETING CERBERUS: DELETE, REMOVE, POP.

_ OK, I lied, only pop isn't as dangerous, but we are.

_ **Pop**: removes item in a index and returns the element.

   _ Pop last: O(1)

   _ Pop other: O(k)

```
>>> rm_list = [1,2,3,2,4,5]
>>> rm_list.pop(3)
    2
>>> print(rm_list)
    [1,2,3,4,5]
>>> rm_list.pop()
    5
```

DANGEROUS USERS: POP

_ **Pop an element from a list**: removes and returns the element at index (default last)

```
function list_pop_impl:
     input: list_object (self), index
     output: element at index if OK, NULL otherwise

     if len(list_object) == 0: return NULL
     if index > len(list_object) or index < 0: return NULL

     element = list_object[index]
     if index == len(list_object):
          return element if list_resize(self, len(list_object)-1)
                         else return NULL
     else:
          // The list might be resized.
          return element if list_ass_slice(index, index+1)
                          else None
```

5.

THE SORTED LIST

SORT VS SORTED

_ **Sorted** will create a new list where its elements will be sorted

```
>>> to_sort_list = [1,3,4,6,5,2]
>>> sorted_list = sorted(to_sort_list)
>>> print(sorted_list)
    [1, 2, 3, 4, 5, 6]
```

_ **Sort** will sort the list in place, saving memory because it doesn't need to create a new list and returns an iterable.

```
>>> to_sort = [1,4,3,2]
>>> to_sort.sort()
>>> print(to_sort)
    [1, 2, 3, 4]
```

```
>>> to_sort = [1,4,3,2]
>>> to_sort.sort(reverse=True)
>>> print(to_sort)
    [4, 3, 2, 1]
```

THE SORTING ALGORITHM: TIMSORT

_ We have our very own sorting algorithm, because we have great developers in the community such as Tim Peters

_ Timsort is an hybrid algorithm, combining:

  _ Merge sort

  _ Insertion sort

_ In a nutshell, the algorithm marches over the array once finding “natural runs” —sorted chunks of the array— and merged intelligently.

_ If the array is less than 64 elements binary insertion sort is used.

_ **sort(*, key=None, reverse=False)**:

  _ Sorting will rely on the < operator of our data

  _ *key* specifies a function of one argument that is used to extract a comparison key from each list element

-1.

BONUS TRACK: BEYOND THE BUILTIN LIST

ARRAY

_ Object type that exposes sequences of basic values: characters, integers, floating point numbers.

```
>>> from array import array
>>> a = array('l', [1, 2, 3])
>>> print(a)
    array('i', [1, 2, 3])
```

MEMVIEW

_ Exposes the C level buffer interface as a Python object.
_ Supports slicing as a subview.

```
>>> from array import array
>>> a = array('l', [1, 2, 3])
>>> print(a)
    array('i', [1, 2, 3])
```

-2.

BONUS TRACK: MORE COOL STUFF

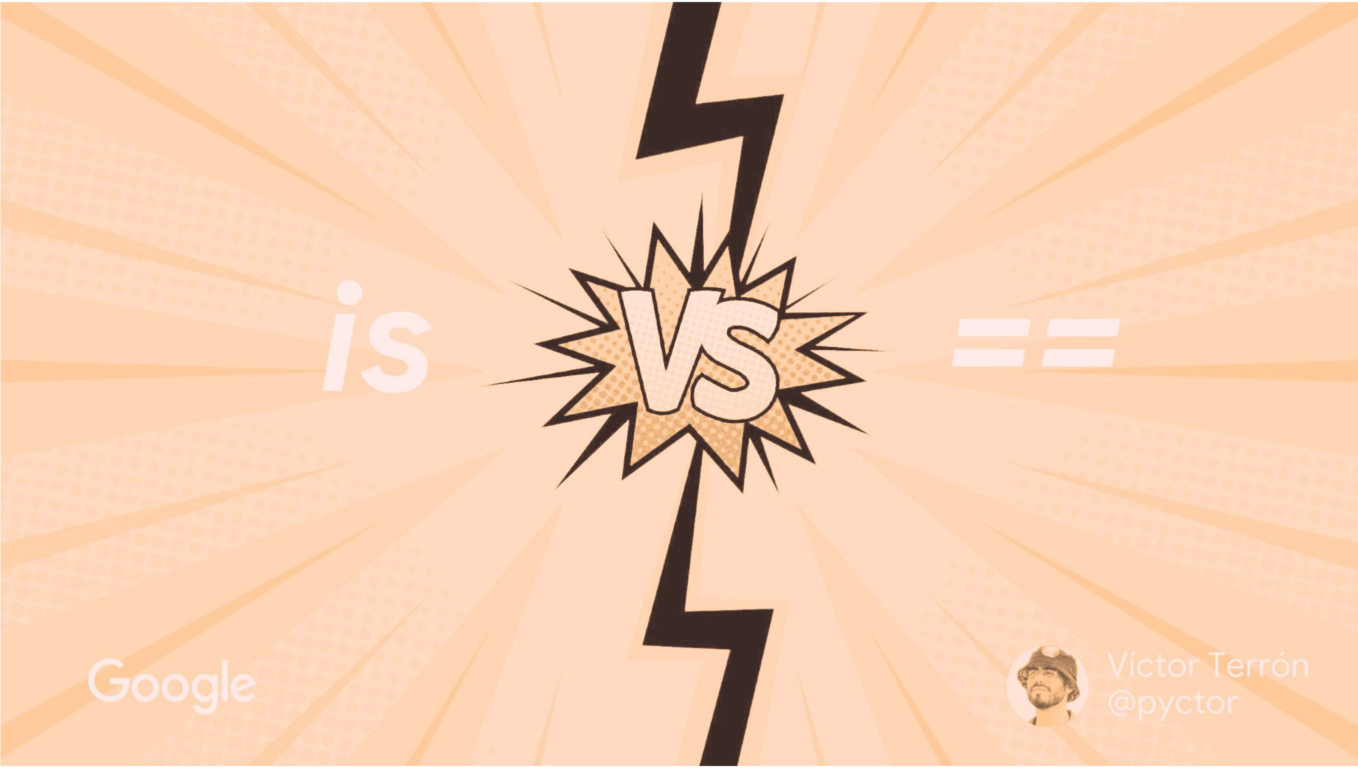
APPEND OR +=

_ Same behaviour, dramatic differences

```
>>> list_add = []
>>> for i in range(5)
      list_add.append(i)
>>> print(list_add)
    [0, 1, 2, 3, 4]
```

```
>>> list_add = []
>>> for i in range(5)
      list_add += [i]
>>> print(list_add)
    [0, 1, 2, 3, 4]
```

HOW TO COMPARE LISTS: == VS IS



Thanks