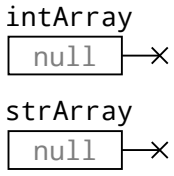
Topic 3.2: Lists

A `list` in Python is similar to an array in statically-typed programming languages such as Java, but are more flexible (more like the Java `ArrayList` class).

Arrays in Java

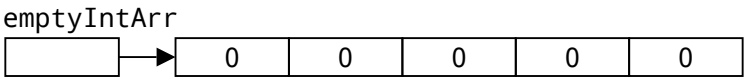
In Java, when declaring an array, we specify the data type that the array stores (which may be a primitive type or an object), followed by square brackets, and then the identifier name.

```
int[] intArray;
String[] strArray;
```

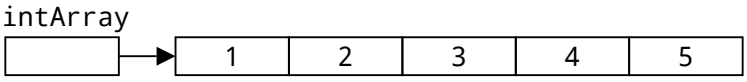


A specific amount of space can be allocated for the array using the `new` keyword, or the array can be initialized to specific values using an array literal, which specifies the values directly within curly braces, `{` and `}`. Examples follow, along with diagrammatic representations of the memory structure created.

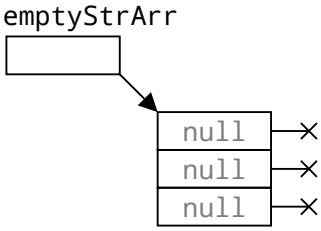
```
int[] emptyIntArr = new int[5];
```



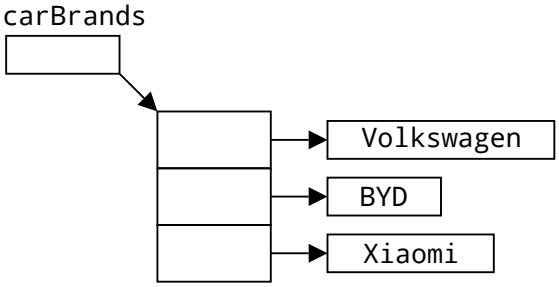

```
int[] intArray = { 1, 2, 3, 4, 5 };
```




```
String[] emptyStrArr = new String[3];
```




```
String[] carBrands = { "Volkswagon", "BYD", "Xiaomi" };
```



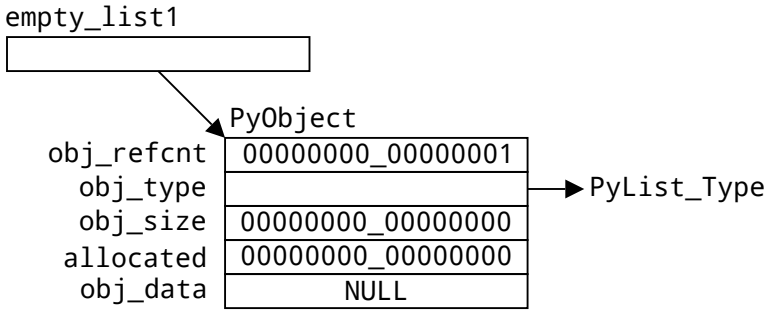
Take note that with an array of Java objects, each array element is the same size, as each element stores the reference to the object. In the given example, the each element of the `String` array stores a reference to a `String` object.

Creating an Empty List in Python

As you surely know by now, every variable in Python is an object, so there is no need to declare the type of a Python variable. Creating a list in Python has some similarities to declaring an array in other languages. The main difference is that allocating memory for the list, including all the complexities of increasing or decreasing the number of elements in the list, is handled by the Python interpreter. This means there is no need to tell Python how many elements will need to be stored in the list.

To create an empty list with no elements in Python, call the list constructor. (Remember all variables in Python are objects, so a list is just another object.)

```
empty_list1 = list()
```



Or, more commonly use the shorthand for creating a list, an empty set of square brackets. (Java uses curly braces for array literals, so note the difference.)

```
empty_list2 = []
```

This statement will create the same memory structure as the previous statement.

Here is some example code that demonstrates the concepts above.

```
>>> import sys
>>> empty_list1 = list()
>>> empty_list1
[]
>>> sys.getsizeof(empty_list1)
56
>>> empty_list2 = []
>>> empty_list2
[]
>>> sys.getsizeof(empty_list2)
56
```

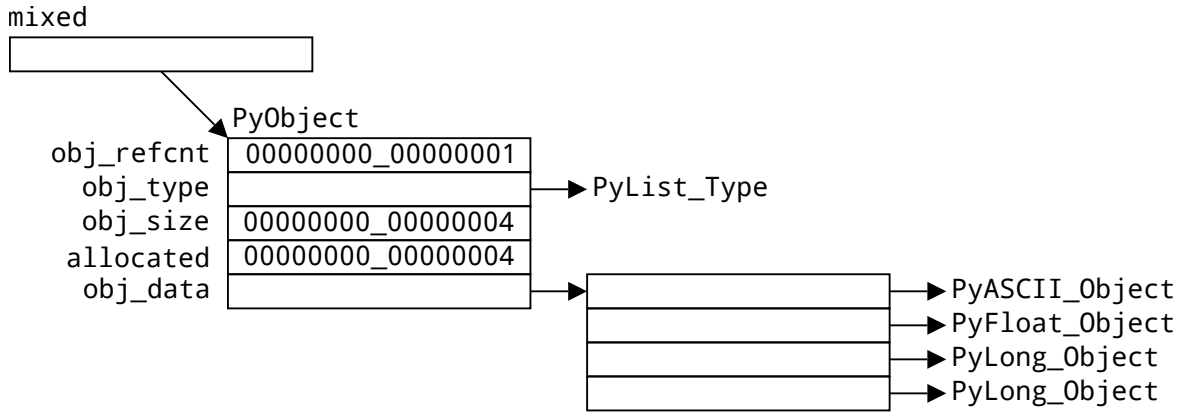
You may notice that the diagram of the memory structure of an empty list suggests that an empty list should only take 40 bytes of memory (Five fields multiplied by 8 bytes per field). That was true of previous versions of Python, but was changed for the 64-bit version as of CPython 3.11. I reiterate: the memory structure diagrams should not be memorized – they are included for a conceptual understanding of a possible way the data structure could be implemented by the underlying interpreter software.

Initializing a List with Elements in Python

Creating a list in Python and initializing it with values is very similar to initializing an array in other languages such as Java. Python uses square brackets to denote a list. Here are some examples.

```
primes = [ 2, 3, 5, 7, 11 ]
car_brands = [ "Volkswagon", "BYD", "Xiaomi" ]
mixed = [ "banana", 1.25, 100, True ]
```

Examine the last example, above. Remember that every variable in Python is an object, so there are no restrictions on the type of each element within any list. Examine the diagrammatic representation of the memory structure used to store the list `mixed`.



You can see `obj_data` is a reference to an array of object references, and each object reference can point to a different data type (a different type of object). It is not an error that the boolean value is stored as a `PyLong_Object`; `True` is simply an integer object storing the value `1` and `False` is an integer object storing the value `0`.

List Length

The `len` function returns the length of a list.

```
>>> primes = [ 2, 3, 5, 7, 11 ]
>>> len(mixed)
4
>>> car_brands = [ "Volkswagon", "BYD", "Xiaomi" ]
>>> len(car_brands)
3
```

Accessing List Elements

Lists use zero-based indexing and are accessed using the index enclosed in square brackets.

```
>>> mixed = [ "banana", 1.25, 100, True ]
>>> mixed[0]
'banana'
>>> mixed[1]
1.25
>>> mixed[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

With four elements, the indexes run from `0` to `3`. Trying to access a non-existent index, such as index `4`, will result in an error.

Python also allows indexing starting from the end of the list by using negative numbers. This counting starts from `1`. For example, using the index `-1` accesses the last element in the list, while using the index `-2` accesses the penultimate element of the list.

```
>>> mixed = [ "banana", 1.25, 100, True ]
>>> mixed[-1]
True
>>> mixed[-4]
'banana'
>>> mixed[-4] is mixed[0]
True
>>> mixed[-5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Mutating Elements

Changing an element is also referred to as mutating it. It is important to get accustomed to this vocabulary because of the importance of immutability and immutable data structures in programming and especially in Python.

Lists are mutable. We can change the values of an element in a list by specifying the element in square brackets (as with accessing the element), but on the left hand side of an assignment operator. For example:

```
mixed[1] = False
```

Will change the value of the second element of the list `mixed` to `False`.

Notice that the type of object may also change. Here is another example.

```
>>> fruits = [ "apple", "banana", "orange" ]
>>> fruits
['apple', 'banana', 'orange']
>>> fruits[2] = "kiwi"
>>> fruits[0] = "peach"
>>> fruits
['peach', 'banana', 'kiwi']
```

Adding More Elements to a List

Unlike static arrays in many other languages, Python dynamically handles growing and shrinking of lists. An additional element can be added to a list using either the `append` method or the `insert` method.

The `append` method adds a number to the end of the list.

```
>>> numbers = [ 1, 2, 3 ]
>>> numbers
[1, 2, 3]
>>> numbers.append(4)
>>> numbers
[1, 2, 3, 4]
```

The `insert` method inserts the element at the given index, shifting the other elements to the right.

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

Removing an Element from a List

An element can be removed from a Python list using either the `remove` method or the `pop` method.

The `remove` method removes the first occurrence of the value from the list. This is determined by checking the equality of values (using `==` comparison). It will remove an object that is determined to be equal, even if it is not the same object (as determined by using the `is` operator).

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

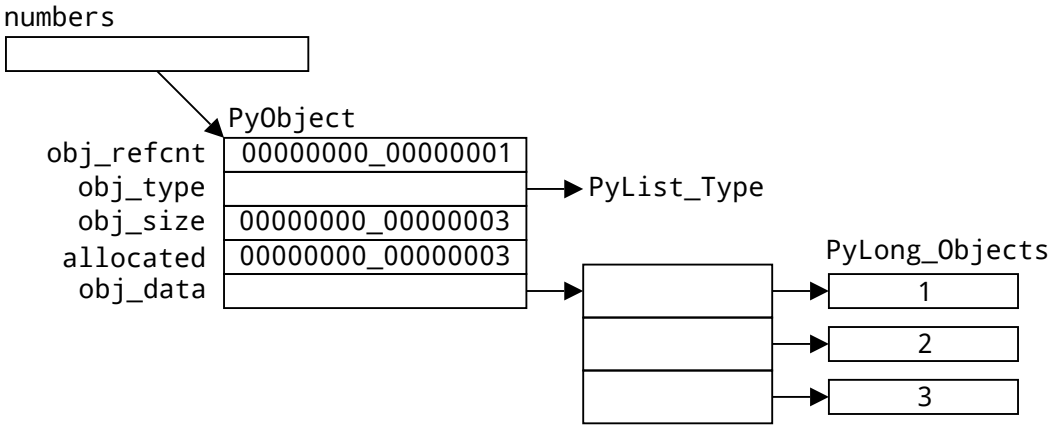
The `pop` method can be used with no parameters to either remove and return the element from the end of the list, or from a specific position in the list by passing the index of the position as a parameter.

```
>>> numbers = [ 1, 2, 3, 4, 5 ]
>>> numbers.pop()
5
>>> numbers
[1, 2, 3, 4]
>>> numbers.append(6)
>>> numbers.pop(1)
2
>>> numbers
[1, 3, 4, 6]
```

Python List Memory Structure

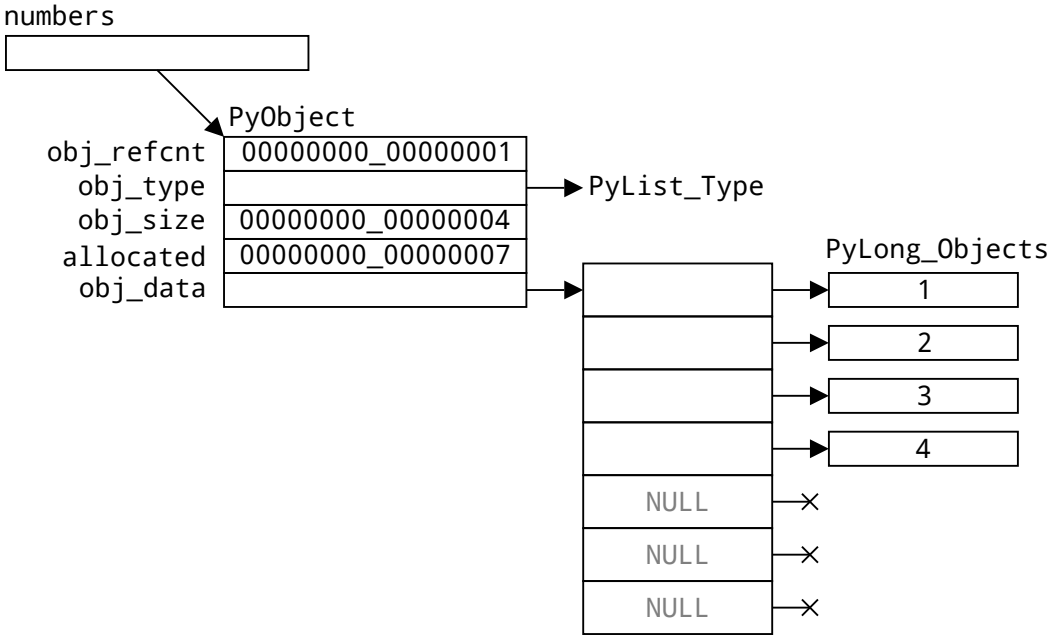
When a list is first created in CPython, it allocates exactly enough space to store the elements. In this case, `obj_size` (the number of elements stored in the list) and `allocated` (the number of elements that storage has been reserved for) will be the same value. For example:

```
numbers = [ 1, 2, 3 ]
```



If code subsequently adds an element to the list, either with the `append` method or `insert` method, or removes an element from the list, with either the `remove` method or the `pop` method, Python resizes the list in an intelligent way.

If we take the above variable, numbers, and append an additional element using the `append` method, there is no space in the `obj_data` array to store new values. A new array of objects must be allocated. However, if we're adding elements to the list, it can be expected it likely that further elements may be added. Allocating memory for a new object array and copying the contents of the current array to the new array is computationally expensive. For this reason, Python allocates space to store elements generously. If an element is added to a list that has only three storage spaces for element references, Python expands this storage to seven.



The `obj_size` keeps track of the number of elements stored in the `list`, while `allocated` keeps track of the number of elements that can be stored in the current `obj_data` array.

The way that Cython calculates how much to expand the list size by is not important, but the formula CPython uses is given below for anyone who might be curious. The `newsize` is the size of the list after addition of elements. It is more computationally expensive to create a new array and copy elements over as the array becomes larger, so CPython becomes more generous in its allocation as the list increases in size.

```
new_allocated = (newsize >> 3) + (newsize < 9 ? 3 : 6) + newsize
```

When removing elements from a list, Python is more conservative and reduces the size of the list when the array is more than half empty (`newsize < (allocated >> 1)`). It still leaves some extra space for new elements to be added, reducing the size to `new_allocated`, as calculated by the previous formula.

List Concatenation (Versus the `append` Method)

It is trivial to concatenate two lists in Python – simply use the `+` operator.

```
>>> list1 = [ "one", 2, 3.0 ]
>>> list2 = list1
>>> id(list1)
4394338368
>>> id(list2)
4394338368
>>> list1 = list1 + [ "two", True ]
>>> list1
['one', 2, 3.0, 'two', True]
>>> id(list1)
4394236992
>>> list2
['one', 2, 3.0]
>>> id(list2)
4394338368
```

As can be seen by the above code, concatenating two lists creates a new list containing the elements of both, rather than modifying the existing list. The original list, still referred to by `list2`, remains unmodified.

```
>>> list1 = [ "one", 2, 3.0 ]
>>> id(list1)
4394299136
>>> list1.append("four")
>>> list1
['one', 2, 3.0, 'four']
>>> id(list1)
4394299136
>>> list2 = [ "one", 2, 3.0 ]
>>> id(list2)
4394236992
>>> list2 = list2 + ["four"]
>>> list2
['one', 2, 3.0, 'four']
>>> id(list2)
4394338368
```

In the example above, using the append method on `list1` kept the same `list` object, and modified the contents of the object, while using concatenation (`+`) on `list2` created a new `list` object and changed the variable `list2` to point to that new object. Considering what was discussed about intelligent allocation for the storage of list elements in obj_data, one should be able to realize

that it is more efficient to use the `append` method to append elements to an existing list rather than using concatenation (`+`) to add an element to a list. List concatenation is used when we wish to add elements to an existing list, but wish to keep the original lists unmodified.

```
>>> workdays = [ "Mon", "Tue", "Wed", "Thu", "Fri" ]
>>> weekend = [ "Sat", "Sun" ]
>>> days_of_the_week = workdays + weekend
>>> workdays
['Mon', 'Tue', 'Wed', 'Thu', 'Fri']
>>> weekend
['Sat', 'Sun']
>>> days_of_the_week
['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
```

Extending a List

The `append` method takes only one parameter – the element to add to the list. It cannot be used to append multiple elements to the existing list.

```
>>> numbers = [ 1, 2, 3 ]
>>> numbers.append(4, 5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list.append() takes exactly one argument (2 given)
>>> numbers
[1, 2, 3]
>>> numbers.append([4, 5])
>>> numbers
[1, 2, 3, [4, 5]]
```

Attempting to pass multiple arguments to the `append` method results in an error, while trying to pass a list of elements to append to the existing list will add the entire list as a single element of the original list. In the example above, the fourth element (index `3`) of the list is set to the list `[4, 5]`, rather than adding two elements, `4` and `5`, to the list.

In order to add a number of elements to the list, there is the extend method. The extend method takes any object that is iterable. An iterable object is any object that can return its elements one at a time, allowing the elements to be looped over in a `for` loop). For now, we will consider only adding the elements of one Python list to another Python list.

The following code shows that with concatenation (`+`), the original list remains unmodified.

```
>>> list1a = [ 1, 2, 3 ]
>>> list1b = list1a
>>> list1a = list1a + [ 4, 5 ]
>>> list1a
[1, 2, 3, 4, 5]
>>> list1b
[1, 2, 3]
```

The following code shows that using the extend method does not create a new `list` object, but rather modifies the existing list. Both `list2a` and `list2b` refer to the same `list` object, so when we append to `list2a`, `list2b` is also modified.

```
>>> list2a = [ 1, 2, 3 ]
>>> list2b = list2a
>>> list2a.extend([4, 5])
>>> list2a
[1, 2, 3, 4, 5]
>>> list2b
[1, 2, 3, 4, 5]
```

Also, be cautious that the `+=` operator acts the same way as the extend method rather than the concatenation operator.

```
>>> list3a = [ 1, 2, 3 ]
>>> list3b = list3a
>>> list3a += [4, 5]
>>> list3a
[1, 2, 3, 4, 5]
>>> list3b
[1, 2, 3, 4, 5]
```

Repetition: The * Operator

The `*` operator creates a list that repeats the elements of the original list.

```
>>> list1 = [ "a", 1 ]
>>> list2 = list1 * 4
>>> list1
['a', 1]
>>> list2
['a', 1, 'a', 1, 'a', 1, 'a', 1]
```

Membership: the `in` Operator

The in operator returns `True` if the element is in the list, otherwise it returns `False`.

```
>>> fruits = ["apple", "banana", "orange"]
>>> print("banana" in fruits)
True
>>> print("grape" in fruits)
False
>>> print("grape" not in fruits)
True
```

List Slicing

List slicing allows extraction of a portion of a list. The syntax follows. The slice includes elements between index `start` and index `end`, including the element at index `start`, but excluding the element at index `end`.

```
list[start:end:step]
```

List slicing creates a new `list`; the original list remains unchanged. Negative indexing is valid, as is a negative `step` value. Examine each of these list slicing expressions carefully.

```
>>> numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> numbers[2:6]
[2, 3, 4, 5]
>>> numbers[:4]
[0, 1, 2, 3]
>>> numbers[7:]
[7, 8, 9]
>>> numbers[-5:-2]
[5, 6, 7]
>>> numbers[1:-1]
[1, 2, 3, 4, 5, 6, 7, 8]
>>> numbers[1:7:2]
[1, 3, 5]
>>> numbers[1:6:2]
[1, 3, 5]
>>> numbers[2:6:-1]
[]
>>> numbers[6:2:-1]
[6, 5, 4, 3]
>>> numbers[::-1]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Notice that if a value for `start` is omitted, it will start at the first element, and if `end` is omitted, the slice will end at the last element. When excluding values, the colon, `:` , is still necessary for Python to determine which value has been excluded.

Other Useful List Methods

Using list slicing to reverse a list, a new `list` object is created. To reverse the list in place (i.e.: without creating a new list), use the **`reverse`** method.

```
>>> fruits = [ "grape", "orange", "apple" ]
>>> fruits[::-1]
['apple', 'orange', 'grape']
>>> fruits = [ "grape", "orange", "apple" ]
>>> fruits.reverse()
>>> fruits
['apple', 'orange', 'grape']
```

The **`sort`** method sorts the list in place (i.e.: it does not create a new `list` object).

```
>>> numbers = [5, 2, 8, 1, 9]
>>> numbers.sort()
>>> numbers
[1, 2, 5, 8, 9]
```

If a variable contains the reference to a list and then it it set to an empty list, the original `list` object is not modified, and the variable is set to refer to a new list. To clear a list in place, use the `clear` method.

```
>>> numbers = [ 1, 2, 3 ]
>>> numbers
[1, 2, 3]
>>> id(numbers)
4427696512
>>> numbers = []
>>> id(numbers)
4427771584
>>> numbers = [ 1, 2, 3 ]
>>> numbers
[1, 2, 3]
>>> id(numbers)
4427696512
>>> numbers.clear()
>>> numbers
[]
>>> id(numbers)
4427696512
```

Nested Lists

Lists can contain lists. It is useful to understand the memory structure of a `list` object to understand why modifying a list contained in another list will result in it appearing as if both have changed.

```
>>> row2 = [ 4, 5, 6 ]
>>> list2d = [ [ 1, 2, 3 ], row2, [ 7, 8, 9 ] ]
>>> list2d
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> row2.append("x")
>>> row2
[4, 5, 6, 'x']
>>> list2d
[[1, 2, 3], [4, 5, 6, 'x'], [7, 8, 9]]
```

A two-dimensional array that has the same length of each row is also called a ***matrix***. An array of three or more dimensions is called a ***tensor***.

```
>>> matrix = [
...     [1, 2, 3],
...     [4, 5, 6],
...     [7, 8, 9]
... ]
>>> matrix
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]

>>> tensor_3d = [
...     [
...         [1, 2, 3],
...         [4, 5, 6]
...     ],
...     [
...         [7, 8, 9],
...         [10, 11, 12]
...     ]
... ]
>>> tensor_3d
[[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]]
```