Python Lists

Table of contents

Comparison 1
Membership 2
Adding elements: `append`, `extend`, concatenation 2
Aliasing 3
Nested lists and matrices 5
Slicing 6
Type hints 8
Unpacking into function arguments 8
Tuples 8

A **list** is a **mutable** ordered collection of elements. These elements are *not necessarily* of the same type. Square brackets `[]` are used to create lists in Python.

```
>>> xs = [1, "apple"]
>>> type(xs)
<class 'list'>
>>> xs[0]
1
>>> xs[0] = 2*xs[1]
>>> xs[0]                  # note: lists are mutable
'appleapple'
```

Comparison

Lists compare point-wise from position zero:

```
>>> [1, 2, 3] < [4, 5, 6]
True
```

```
>>> [7, 2, 3] < [4, 5, 6]
False
>>> [] < [1]
True
```

The second line is the one to read carefully: the comparison stops at position zero, where `7 < 4` is `False`, and never looks at positions one and two.

Membership

```
>>> 1 in [1, 2, 3]
True
>>> 0 in [1, 2, 3]
False
>>> 123 in [1, 2, 3]
False
>>> [1] in [1, 2, 3]
False
```

`in` tests for an *element*, not a sub-list — the last line is `False` because `[1, 2, 3]` contains the integer `1`, not the list `[1]`.

Adding elements: `append`, `extend`, concatenation

`xs + [y]` **creates a new list**, which must be reassigned back to `xs` to have an effect:

```
>>> xs = [0, 1, 2]
>>> xs = xs + [3]
>>> xs
[0, 1, 2, 3]              # "this is the original list updated"
```

Read the comment as a statement about the *value* of `xs`, not about the object: `xs = xs + [...]` builds a **new** list and rebinds `xs` to it (see the aliasing transcript below), which is exactly why it breaks an alias where `append` does not.

Alternatively, using a method that `list` provides — `xs.append(y)` mutates `xs` **in place**:

```
>>> xs = [0, 1, 2]
>>> xs.append(3)
>>> xs
[0, 1, 2, 3]
```

`append` adds its argument as a *single new element*, even when that argument is itself a list:

```
>>> xs = [1, 2, 3]
>>> xs.append([4,5])
>>> xs
[1, 2, 3, [4, 5]]
```

`xs.extend(ys)` instead appends every element of `ys` individually — code similar to `xs = xs + ys`:

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

Aliasing

Take care when (trying to) copy lists. Assigning one list variable to another does **not** copy it — both names refer to the *same* list object:

```
>>> xs = [1, 2, 3]
>>> ys = xs
>>> ys[-1] = 9
>>> ys
[1, 2, 9]
>>> xs
[1, 2, 9]
```

Whether an operation preserves this aliasing relationship depends on whether it mutates the list in place or creates a new one:

```
>>> xs = [1, 2, 3]
>>> ys = xs           # ys is an alias of xs
>>> xs.append(4)      # appending to a list
>>> ys
[1, 2, 3, 4]
>>> xs += [5]         # appending to a list
>>> ys
[1, 2, 3, 4, 5]
>>> xs = xs + [6]     # concatenating two lists
>>> ys
[1, 2, 3, 4, 5]       # aliasing relationship broken!
```

The trap is that `xs += [5]` and `xs = xs + [6]` look like the same operation and are not: the first mutates the object both names share, the second builds a *new* object and rebinds only `xs`.

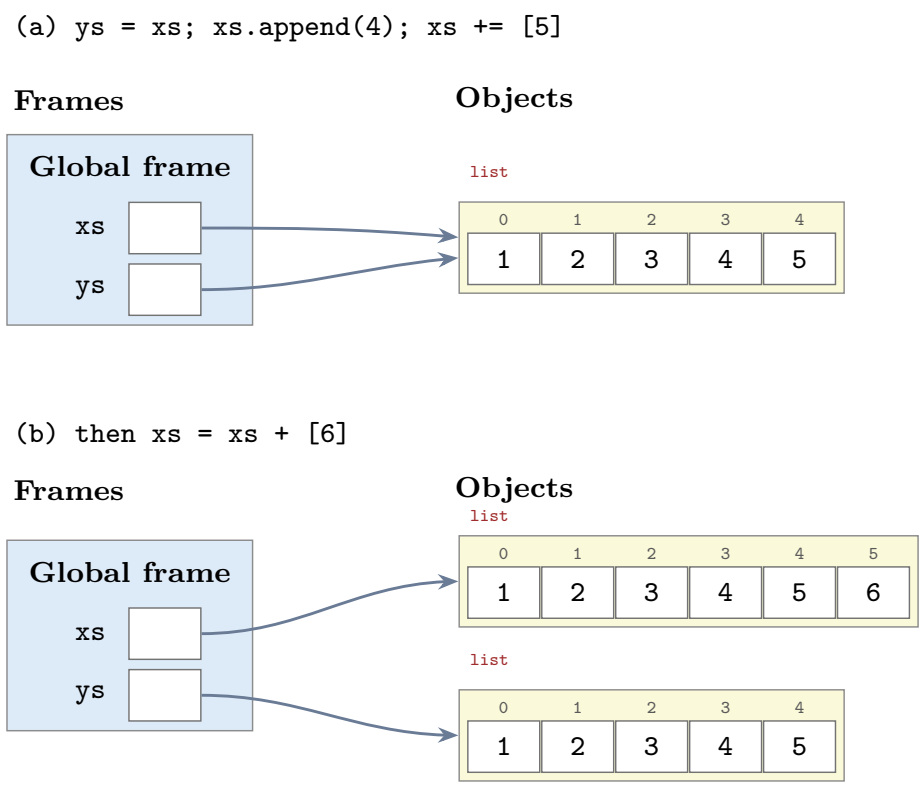

Figure 1: What the transcript above is doing to memory. In (a) `xs` and `ys` are two names for one list object, so the in-place operations `append` and `+=` are visible through both. In (b) `xs = xs + [6]` builds a second list object and rebinds `xs` to it; `ys` still refers to the original, which is what *aliasing relationship broken* means. The layout is Python Tutor's: frames on the left in pale blue, objects on the right in pale yellow, a reference as an arrow, and aliasing as two arrows converging on one object.

`.copy()` breaks the alias for a flat list:

```
>>> xs = [1, 2, 3]
>>> ys = xs.copy()
>>> ys[-1] = 9
>>> xs
[1, 2, 3]
>>> ys
[1, 2, 9]
```

Passing lists to functions

A more nuanced case is when passing lists (but really *list references*) to functions — mutating a parameter inside a function mutates the caller's list too:

```
>>> def foo(ys):
...     ys[0] = -100
>>> xs = [1, 2, 3]
>>> foo(xs)
>>> xs
[-100, 2, 3]
```

This is the same picture as (a) in Figure 1, with `ys` being a name in `foo`'s frame rather than the global one.

Nested lists and matrices

A list can have another list as an element:

```
>>> xs = [ [1,2], [5,6,7] ]
>>> len(xs)
2
>>> len(xs[-1])
3
```

`len` counts the *outer* elements — two, not five.

A **matrix** can therefore be represented by a list-of-lists in Python:

$$\mathbb{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \equiv \texttt{[[1, 2, 3], [4, 5, 6], [7, 8, 9]]}$$

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

```
6
>>> ass[1,-1]
TypeError: list indices must be integers or slices, not tuple
```

The last line is the one worth remembering: a nested list is indexed with *repeated* brackets, `ass[1][-1]`. The comma form `ass[1,-1]` is a tuple and is a `TypeError` for a list.

Deep copying

`.copy()` only performs a **shallow copy** — nested mutable elements (like inner lists) are still shared between the original and the copy:

```
>>> ass = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> bss = ass.copy()
>>> bss[0][0] = 0
>>> bss
[[0, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> ass
[[0, 2, 3], [4, 5, 6], [7, 8, 9]]   # A was changed!
```

A true independent copy of nested structures needs a **deep copy**, which is *not* built into `list` — see `copy.deepcopy`.

Slicing

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

The three fields are `start:stop:step` with `stop` exclusive: `xs[3:6]` takes positions 3, 4 and 5; `xs[::2]` takes every second element of the whole list; and a negative step makes `xs[7:2:-2]` walk *backwards* from position 7, stopping before position 2.

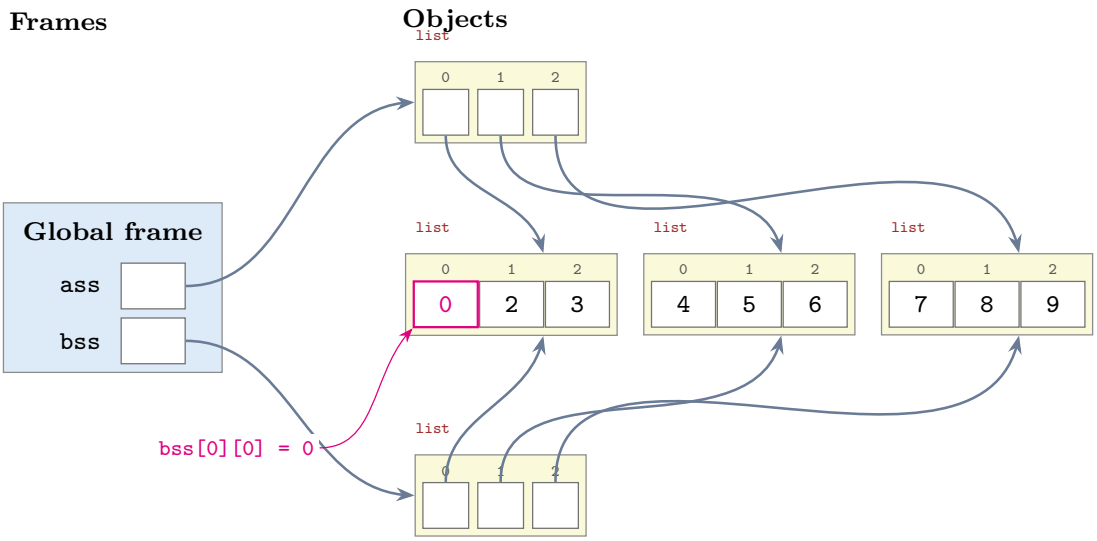

Figure 2: Why `ass` changed. `bss = ass.copy()` made a genuinely new *outer* list — that is why `bss` and `ass` are two distinct objects — but it copied the outer list's three references, not the objects they point at, so both outer lists share the same three inner lists. Writing `bss[0][0] = 0` reaches through a shared reference and the change shows up through `ass` as well. For this to work as expected we need a *deep-copy*, which is not built-in — see `copy.deepcopy`. Magenta marks the mutated cell.

Type hints

> **Warning.** Type-hinting lists (e.g. `list[int]`) is a new feature in Python 3.9 and greater.

Unpacking into function arguments

It is possible to "unbracket" a list when passing it to functions, with `*`:

```
>>> def f(x, y, z):
...     return x + y + z
>>> x = [1, 2, 3]
>>> f(x)
TypeError: f() missing 2 required positional arguments: 'y' and 'z'
>>> f(*x)
6
```

Tuples

Tuples are **immutable ordered collections** whereas lists are **mutable ordered collections**. Both can be sliced and indexed. See python-primitive-data-types[1]; a tuple's immutability is what makes it usable as a dictionary key, where a list is not — see python-dictionaries[2].

[1] courses/csse1001/python-primitive-data-types.pdf
[2] courses/csse1001/python-dictionaries.pdf