Python Lists

Table of contents

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

```
>>> xs = [1, "apple"]
>>> type(xs)
<class 'list'>
>>> xs[0]
1
>>> xs[0] = 2*xs[1]
>>> xs[0]
'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
```

Membership

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

Adding elements: `append` vs. concatenation

`xs.append(y)` mutates `xs` **in place**, adding `y` as a single new element (even if `y` is itself a list):

```
>>> xs = [0, 1, 2]
>>> xs.append(3)
>>> xs
[0, 1, 2, 3]
>>> xs.append([4, 5])
>>> xs
[0, 1, 2, 3, [4, 5]]
```

`xs + [y]` instead **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]
```

`xs.extend(ys)` mutates `xs` in place, appending every element of `ys`:

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

Aliasing

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)      # mutates in place
>>> ys
[1, 2, 3, 4]
>>> xs += [5]         # also mutates in place
>>> ys
[1, 2, 3, 4, 5]
>>> xs = xs + [6]     # creates a NEW list, only rebinds xs
>>> ys
[1, 2, 3, 4, 5]        # aliasing relationship broken!
```

`.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

Lists are passed to functions by reference, so 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]
```

Nested lists and matrices

A list can contain another list as an element, e.g. representing a matrix as a list-of-lists:

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

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

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]]   # ass was changed too!
```

A true independent copy of nested structures needs a **deep copy**, which isn't 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]
```

Type hints

Type-hinting the elements of a list (e.g. `list[int]`) is a feature new in Python 3.9 and greater.

Unpacking into function arguments

A list can be "unbracketed" into positional arguments with `*`:

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