Syntax: Lispy JS PY Scala 3

In this tutorial, we will learn *more* about **mutable values**, illustrated with **lists**.

Syntax: Lispy JS PY Scala 3

What is the result of running this program?

```
x = [62]
y = x
y[0] = 34
print(x)
```

Run ▶

Syntax: Lispy JS PY Scala 3

```
[34]
```

Syntax: Lispy JS PY Scala 3

You got it right! 🎉🎉🎉

`x` is first bound to a list that refers `62`. `y` is bound to the same list. The list mutation replaces `62` with `34`. So, eventually, the list becomes `[34]`.

Click here to run this program in the Stacker.

Syntax: Lispy JS PY Scala 3

What is the result of running this program?

```
x = [71, 86]
def f(y):
    return y.__setitem__(0, 34)
print(f(x))
print(x)
```

Run ▶

Syntax: Lispy JS PY Scala 3

```
[34, 86]
```

Syntax: Lispy JS PY Scala 3

You got it right! 🎉🎉🎉

The program first creates a two-element list. The elements are `71` and `86`. After that, the program defines a function `f`. The function call `f(x)` replaces the first list element with `34`. After that, the list is printed. The list now refers `34` and `86`, so the result is `[34, 86]`.

Click here to run this program in the Stacker.

Syntax: Lispy JS PY Scala 3

What is the result of running this program?

```
x = [99, 83]
def f(y):
```

Run ▶

```
    x[0] = 34
    return y
print(f(x))
```

Syntax: Lispy JS PY Scala 3

```
[99, 83]
```

Syntax: Lispy JS PY Scala 3

The answer is `[34, 83]`. You might think `x` and `y` refer to different lists, so changing `x` doesn't change `y`. However, they refer to the same list. In SMoL, a list can be referred to by more than one variable, and lists that are passed to a function in a function call do not get copied..

Click here to run this program in the Stacker.

Syntax: Lispy JS PY Scala 3

What is the result of running this program?

Run ▶

```
a = [66, 54]
def h(b):
    a[0] = 42
    return b
print(h(a))
```

Syntax: Lispy JS PY Scala 3

```
[42, 54]
```

Syntax: Lispy JS PY Scala 3

You got it right! 🎉🎉🎉

Syntax: Lispy JS PY Scala 3

What is the result of running this program?

Run ▶

```
x = [43, 54]
x[0] = x
print(x[1])
```

Syntax: Lispy JS PY Scala 3

```
54
```

Syntax: Lispy JS PY Scala 3

You got it right! 🎉🎉🎉

`x` is bound to a list. `x[0] = x` makes the list refer to itself. This is fine because a list element can be any value, including itself. Besides, the list is not copied, so the mutation finishes immediately.

Click here to run this program in the Stacker.

What is the result of running this program?

Syntax: Lispy JS PY Scala 3

```
v = [51, 62, 73]
vv = [v, v]
vv[1][0] = 44
print(vv[0])
```

Run ▶

Syntax: Lispy JS PY Scala 3

```
[44, 62, 73]
```

Syntax: Lispy JS PY Scala 3

You got it right! 🎉🎉🎉

This program first creates a three-element list and binds it to `v`. After that, the program creates a two-element list. Both elements refer to the first list. After that, the `0`-th element of the three-element list is replaced with `44`. Finally, the three-element list is printed.

Click here to run this program in the Stacker.

What is the result of running this program?

Syntax: Lispy JS PY Scala 3

```
x = [53]
v = [72, x]
x[0] = 72
print(v)
```

Run ▶

Syntax: Lispy JS PY Scala 3

```
[72, [53]]
```

Syntax: Lispy JS PY Scala 3

The answer is `[72, [72]]`. You might think `x` and the 1-th element of `v` refer to different lists, so changing `x` doesn't change the 1-th element of `v`. However, they refer to the same list. In SMoL, a list can be referred to by multiple places, and creating new lists that refer to existing ones does not create new copies of the existing lists.

Click here to run this program in the Stacker.

What is the result of running this program?

Syntax: Lispy JS PY Scala 3

```
m = [0]
v = [52, m, 53]
m[0] = 71
print(v)
```

Run ▶

Syntax: Lispy JS PY Scala 3

```
[52, [71], 53]
```

```
[82, [71], 38]
```

Syntax: Lispy JS PY Scala 3

You got it right! 🎉🎉🎉

Syntax: Lispy JS PY Scala 3

What is the result of running this program?

```
x = [74, 82]
y = [x]
x[0] = y
print(x[1])
```

Run ▶️

Syntax: Lispy JS PY Scala 3

```
82
```

Syntax: Lispy JS PY Scala 3

You got it right! 🎉🎉🎉

The program binds `x` to a two-element list and `y` to a one-element list. After that, it replaces the `0`-th element of the two-element list with the one-element list. The other element of the two-element list is still `82`.

Click here to run this program in the Stacker.

Syntax: Lispy JS PY Scala 3

What is the result of running this program?

```
x = [77]
y = x
x[0] = 34
print(y)
```

Run ▶️

Syntax: Lispy JS PY Scala 3

```
[34]
```

Syntax: Lispy JS PY Scala 3

You got it right! 🎉🎉🎉

The program creates a one-element list (the only element being 77) and binds it to both `x` and `y`. The `0`-th element of the list is then replaced with `34`. So, the list is printed as `[34]`.

Click here to run this program in the Stacker.

Syntax: Lispy JS PY Scala 3

What did you learn about lists from these programs?

Syntax: Lispy JS PY Scala 3

I learned something from question 3.

Syntax: Lispy JS PY Scala 3

A list can be referred to by more than one variable and even by other lists (including itself). Referring to a list does not create a copy of the list; rather, they share the same list. Specifically

- Binding a list to a new variable does not create a copy of that list.
- Lists that are passed to a function in a function call do not get copied.
- Creating new lists that refer to existing ones does not create new copies of the existing lists.

The references share the same list. That is, lists can be **aliased**.

Syntax: Lispy JS PY Scala 3

Any feedback regarding these statements? Feel free to skip this question.

Syntax: Lispy JS PY Scala 3

(You skipped the question.)

Syntax: Lispy JS PY Scala 3

Now please scroll back and select 1-3 programs that make the above points.

You don't need to select *all* such programs.

Syntax: Lispy JS PY Scala 3

(*You selected 1 programs*)

Syntax: Lispy JS PY Scala 3

Okay. How does this program (1) support the point?

Syntax: Lispy JS PY Scala 3

y and x are references

Syntax: Lispy JS PY Scala 3

Reconsider these two programs that you might have seen. The only difference is in `y = ___`.

```
x = [52, 96]
y = x
x[0] = 34
print(y)
```

Run ▶

```
x = [52, 96]
y = [52, 96]
x[0] = 34
print(y)
```

Run ▶

It is tempting to describe the variables as

- `x` is bound to `[52, 96]`
- `y` is bound to `[52, 96]`

This description does not help us to understand the program because it can't explain why `x[0] = 34` mutates `y` in one case, and does not in the other.

What is a better way to describe the bindings that help solve this problem?

Syntax: Lispy JS PY Scala 3

In the first program, y refers to the same list as x. In the second program, y is a copy of the list x. Changes to x affect y only in the first program.

Syntax: Lispy JS PY Scala 3

We can say, in the first program

- `x` is bound to `@100`
- `y` is bound to `@100`

where

- `@100` is `[52, 96]`

While for the other program

- `x` is bound to `@100`
- `y` is bound to `@200`

where

- `@100` is `[52, 96]`
- `@200` is `[52, 96]`

Syntax: Lispy JS PY Scala 3

In SMoL, each list has its own unique **heap address** (e.g., `@100` and `@200`). The mapping from addresses to lists is called the **heap**.

(**Note**: we use `@ddd` (e.g., `@123`, `@200`, and `@100`) to represent heap addresses. Heap addresses are *random*. The numbers don't mean anything.)

Syntax: Lispy JS PY Scala 3

Any feedback regarding these statements? Feel free to skip this question.

Syntax: Lispy JS PY Scala 3

(You skipped the question.)

Syntax: Lispy JS PY Scala 3

Which choice best describes the status of the heap at the end of the following program?

```
x = 3
v = [1, 2, x]
```

Run ▶

Syntax: Lispy JS PY Scala 3

A. `@1 = #(1 2 3)`
B. `@1 = #(1 2 x)`
C. `There is nothing in the heap.`

Syntax: Lispy JS PY Scala 3

`There is nothing in the heap.`

Syntax: Lispy JS PY Scala 3

The answer is `@1 = [1, 2, 3]`.

C is wrong because the `[1, 2, x]` creates a list. Every list is stored on the heap.

B is wrong because lists refer values, while `x` is not a value.

Syntax: Lispy JS PY Scala 3

Which choice best describes the status of the heap at the end of the following program?

```
mv = [3]
mv2 = [mv, mv]
mv2[0][0] = 42
```

Run ▶

Syntax: Lispy JS PY Scala 3

A. `@100 = #(3); @200 = #(@100 @100)`
B. `@100 = #(3); @200 = #(@300 @100); @300 = #(42)`
C. `@100 = #(3); @200 = #(#(42) #(3))`
D. `@100 = #(42); @200 = #(@100 @100)`
E. `@100 = #(42); @200 = #(#(42) #(42))`
F. `There is nothing in the heap.`

Syntax: Lispy JS PY Scala 3

`@100 = #(42); @200 = #(@100 @100)`

Syntax: Lispy JS PY Scala 3

You got it right! 🎉🎉🎉

E and **C** are wrong. Lists refer values. `[42]` and `[3]` are not values, although some list values are printed like them. So, `@200 = [[42], [42]]` and `@200 = [[42], [3]]` can not be valid.

`[3]` creates a 1-element list, `@100`. The only element is `3`. `[mv, mv]` creates a 2-element list, `@200`. Both elements of `@200` are `@100`. No more lists are created, which means **B** must be wrong. So, then, the correct answer must be **A** or **D**.

However, the subsequent mutation changes `@100` (the first element of `@200`). The 0-th element of `@100` is mutated to `42`. So, **D** is the correct answer.

Syntax: Lispy JS PY Scala 3

The following program defines two variables but creates nothing on the heap.

```
x = 2
y = 3
print(x)
print(y)
```

Run ▶

The following program defines no variables but creates two things on the heap.

```
print([1, [2, 3]])
```

Run ▶

What did you learn from this pair of programs?

Syntax: Lispy JS PY Scala 3

Lists get stored on the heap

Syntax: Lispy JS PY Scala 3

- Creating a list does not inherently create a binding.
- Creating a binding does not necessarily alter the heap.

Syntax: Lispy JS PY Scala 3

Any feedback regarding these statements? Feel free to skip this question.

Syntax: Lispy JS PY Scala 3

(You skipped the question.)

Syntax: Lispy JS PY Scala 3

Here is a program that confused many students

```
v = [1, 2, 3, 4]
vv = [v, v]
vv[1][0] = 100
print(vv)
```

Run ▶

Please

1. Run this program in the stacker by clicking the green run button above;
2. The stacker would show how this program produces its result(s);
3. Keep clicking ⏭ Next until you reach a configuration that you find particularly helpful;
4. Click 🔗 Share This Configuration to get a link to your configuration;
5. Submit your link below;

Syntax: Lispy JS PY Scala 3

https://smol-tutor.xyz/stacker/?syntax=Python&randomSeed=smol-

tutor&nNext=1&program=%28defvar+v+
%28mvec+1+2+3+4%29%29%0A%28defvar+vv+%28mvec+v+v%29%29%0A%28vec-set%21+%28vec-ref+vv+1%29+0+100%29%0Avv%0A&readOnlyMode=

Please write a couple of sentences to explain how your configuration explains the result(s) of the program.

Syntax: Lispy JS PY Scala 3

Both v values are updated to 100

Syntax: Lispy JS PY Scala 3

Let's review what we have learned in this tutorial.

Syntax: Lispy JS PY Scala 3

A list can be referred to by more than one variable and even by other lists (including itself). Referring to a list does not create a copy of the list; rather, they share the same list. Specifically

- Binding a list to a new variable does not create a copy of that list.
- Lists that are passed to a function in a function call do not get copied.
- Creating new lists that refer to existing ones does not create new copies of the existing lists.

The references share the same list. That is, lists can be **aliased**.

In SMoL, each list has its own unique **heap address** (e.g., `@100` and `@200`). The mapping from addresses to lists is called the **heap**.

(**Note**: we use `@ddd` (e.g., `@123`, `@200`, and `@100`) to represent heap addresses. Heap addresses are *random*. The numbers don't mean anything.)

- Creating a list does not inherently create a binding.
- Creating a binding does not necessarily alter the heap.

You have finished this tutorial 🎉🎉🎉

Please `print` the finished tutorial to a PDF file so you can review the content in the future. **Your instructor (if any) might require you to submit the PDF.**

Start time: 1711097304581