Lists are Mutable

Unlike strings, lists are **mutable**. This means we can change an item in a list by accessing it directly as part of the assignment statement. Using the indexing operator (square brackets) on the left side of an assignment, we can update one of the list items.

```python
fruit = ["banana", "apple", "cherry"]
print(fruit)

fruit[0] = "pear"
fruit[-1] = "orange"
print(fruit)

```

ActiveCode: 1 (ch09_7)

Run

```
['banana', 'apple', 'cherry']
['pear', 'apple', 'orange']
```

An assignment to an element of a list is called **item assignment**. Item assignment does not work for strings. Recall that strings are immutable.

Here is the same example in codelens so that you can step thru the statements and see the changes to the list elements.

```python
fruit = ["banana", "apple", "cherry"]

fruit[0] = "pear"
fruit[-1] = "orange"
```

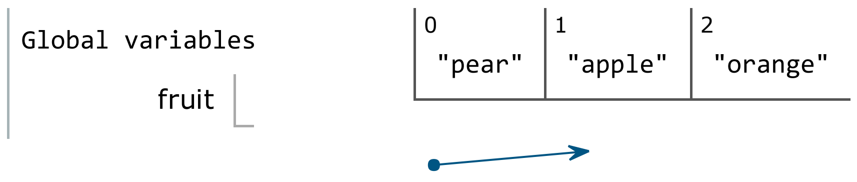
<< First   < Back   Program terminated   Forward >   Last >>

➡ line that has just executed

➡ next line to execute

Frames   Objects

list



CodeLens: 1 (item_assign)

By combining assignment with the slice operator we can update several elements at once.

```python
alist = ['a', 'b', 'c', 'd', 'e', 'f']
alist[1:3] = ['x', 'y']
print(alist)

```

ActiveCode: 2 (ch09_8)

Run

```
['a', 'x', 'y', 'd', 'e', 'f']
```

We can also remove elements from a list by assigning the empty list to them.

```python
alist = ['a', 'b', 'c', 'd', 'e', 'f']
alist[1:3] = []
print(alist)

```

ActiveCode: 3 (ch09_9)

Run

```
['a', 'd', 'e', 'f']
```

We can even insert elements into a list by squeezing them into an empty slice at the desired location.

```python
alist = ['a', 'd', 'f']
alist[1:1] = ['b', 'c']
print(alist)
alist[4:4] = ['e']
print(alist)

```

ActiveCode: 4 (ch09_10)

Run

```
['a', 'b', 'c', 'd', 'f']
['a', 'b', 'c', 'd', 'e', 'f']
```

Check your understanding

List Deletion

Using slices to delete list elements can be awkward and therefore error-prone. Python provides an alternative that is more readable. The `del` statement removes an element from a list by using its position.

```python
a = ['one', 'two', 'three']
del a[1]
print(a)

alist = ['a', 'b', 'c', 'd', 'e', 'f']
del alist[1:5]
print(alist)

```

ActiveCode: 5 (ch09_11)

Run

```
['one', 'three']
['a', 'f']
```

As you might expect, `del` handles negative indices and causes a runtime error if the index is out of range. In addition, you can use a slice as an index for `del`. As usual, slices select all the elements up to, but not including, the second index.

Note

This workspace is provided for your convenience. You can use this activecode window to try out anything you like.

```


```

ActiveCode: 6 (scratch_09_01)

Run

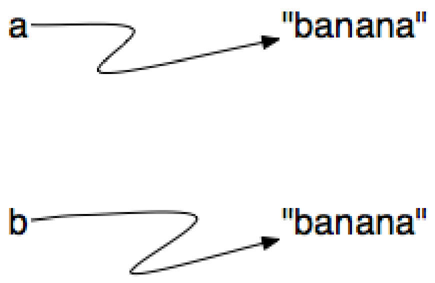
Objects and References

If we execute these assignment statements,

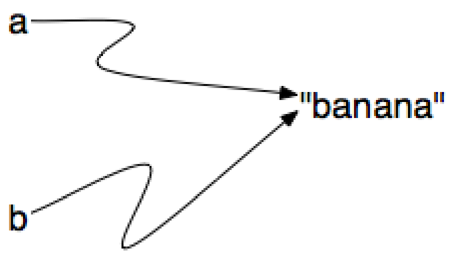
```
a = "banana"
b = "banana"
```

we know that `a` and `b` will refer to a string with the letters `"banana"` . But we don't know yet whether they point to the *same* string.

There are two possible ways the Python interpreter could arrange its internal states:



or



In one case, `a` and `b` refer to two different string objects that have the same value. In the second case, they refer to the same object. Remember that an object is something a variable can refer to.

We already know that objects can be identified using their unique identifier. We can also test whether two names refer to the same object using the *is* operator. The *is* operator will return true if the two references are to the same object. In other words, the references are the same. Try our example from above.

```
a = "banana"
b = "banana"

print(a is b)

```

ActiveCode: 7 (chp09_is1)

Run

```
True
```

The answer is `True`. This tells us that both `a` and `b` refer to the same object, and that it is the second of the two reference diagrams that describes the relationship. Since strings are *immutable*, Python optimizes resources by making two names that refer to the same string value refer to the same object.

This is not the case with lists. Consider the following example. Here, `a` and `b` refer to two different lists, each of which happens to have the same element values.

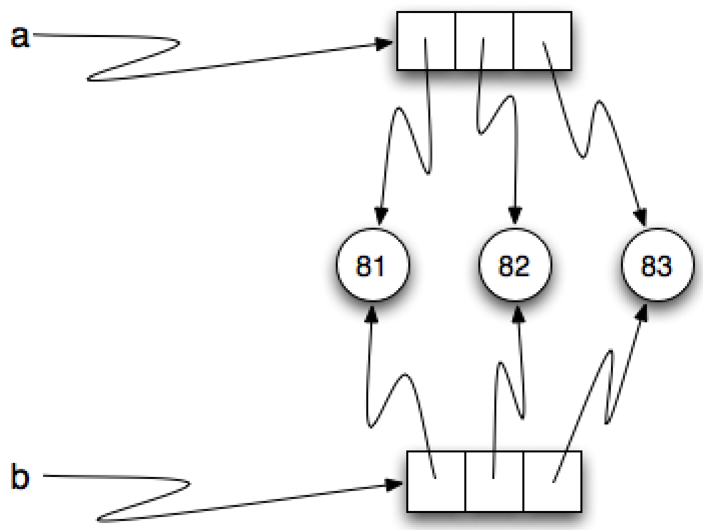
```
a = [81,82,83]
b = [81,82,83]

print(a is b)

print(a == b)

```

ActiveCode: 8 (chp09_is2)

Run

```
False
True
```

The reference diagram for this example looks like this:



`a` and `b` have the same value but do not refer to the same object.

There is one other important thing to notice about this reference diagram. The variable `a` is a reference to a **collection of references**. Those references actually refer to the integer values in the list. In other words, a list is a collection of references to objects. Interestingly, even though `a` and `b` are two different lists (two different collections of references), the integer object `81` is shared by both. Like strings, integers are also immutable so Python optimizes and lets everyone share the same object.

Here is the example in codelens. Pay particular attention to the id values.

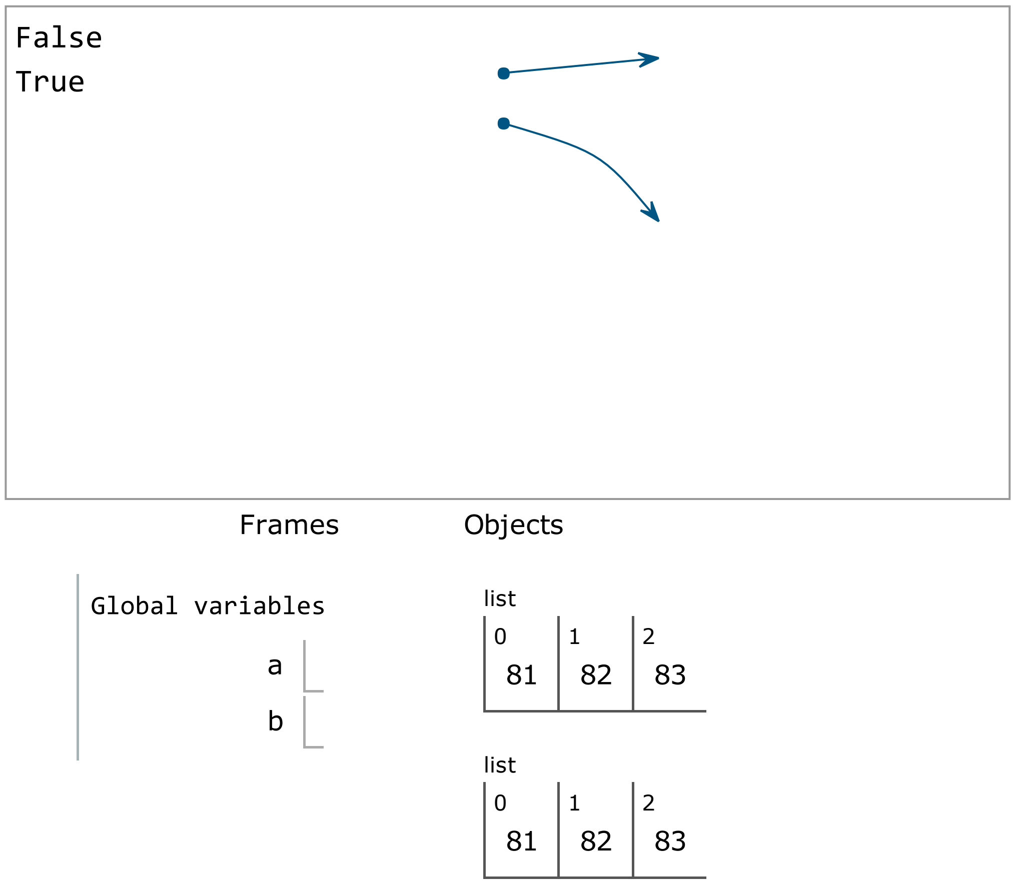
```
a = [81,82,83]
b = [81,82,83]

print(a is b)
print(a == b)
```

<< First   < Back   Program terminated   Forward >   Last >>

➡ line that has just executed

➡ next line to execute

Program output:



```
False
True
```

CodeLens: 2 (chp09_istrace)

Aliasing

Since variables refer to objects, if we assign one variable to another, both variables refer to the same object:

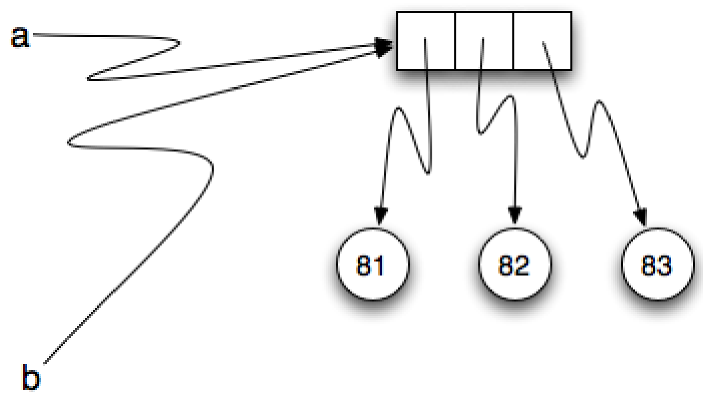
```python
a = [81, 82, 83]
b = a
print(a is b)

```

ActiveCode: 9 (listalias1)

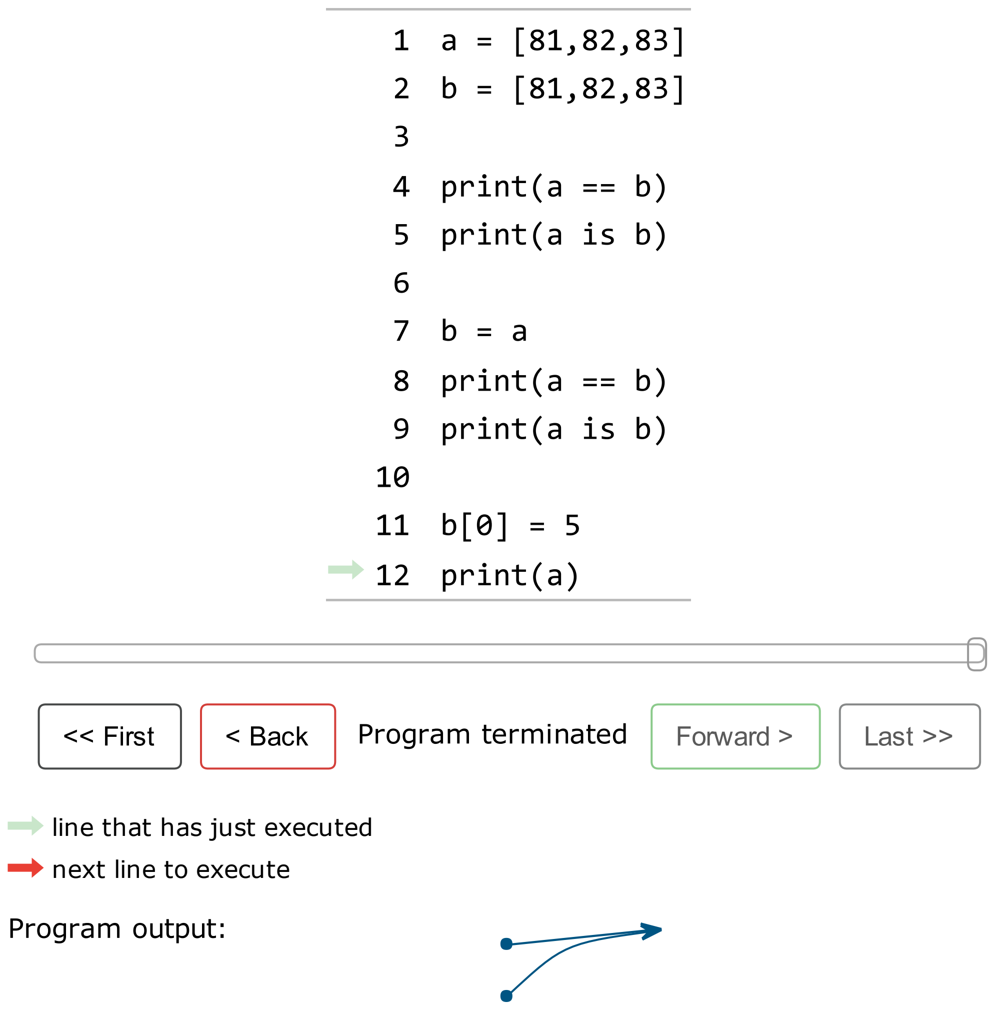
Run

```
True
```

In this case, the reference diagram looks like this:



Because the same list has two different names, `a` and `b`, we say that it is **aliased**. Changes made with one alias affect the other. In the codelens example below, you can see that `a` and `b` refer to the same list after executing the assignment statement `b = a`.



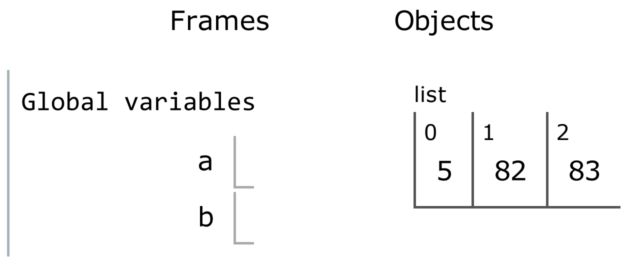
```
a = [81,82,83]
b = [81,82,83]

print(a == b)
print(a is b)

b = a
print(a == b)
print(a is b)

b[0] = 5
print(a)
```

➡ line that has just executed

➡ next line to execute

Program output:

```
True
False
True
True
[5, 82, 83]
```



CodeLens: 3 (chp09_is3)

Although this behavior can be useful, it is sometimes unexpected or undesirable. In general, it is safer to avoid aliasing when you are working with mutable objects. Of course, for immutable objects, there’s no problem. That’s why Python is free to alias strings and integers when it sees an opportunity to economize.

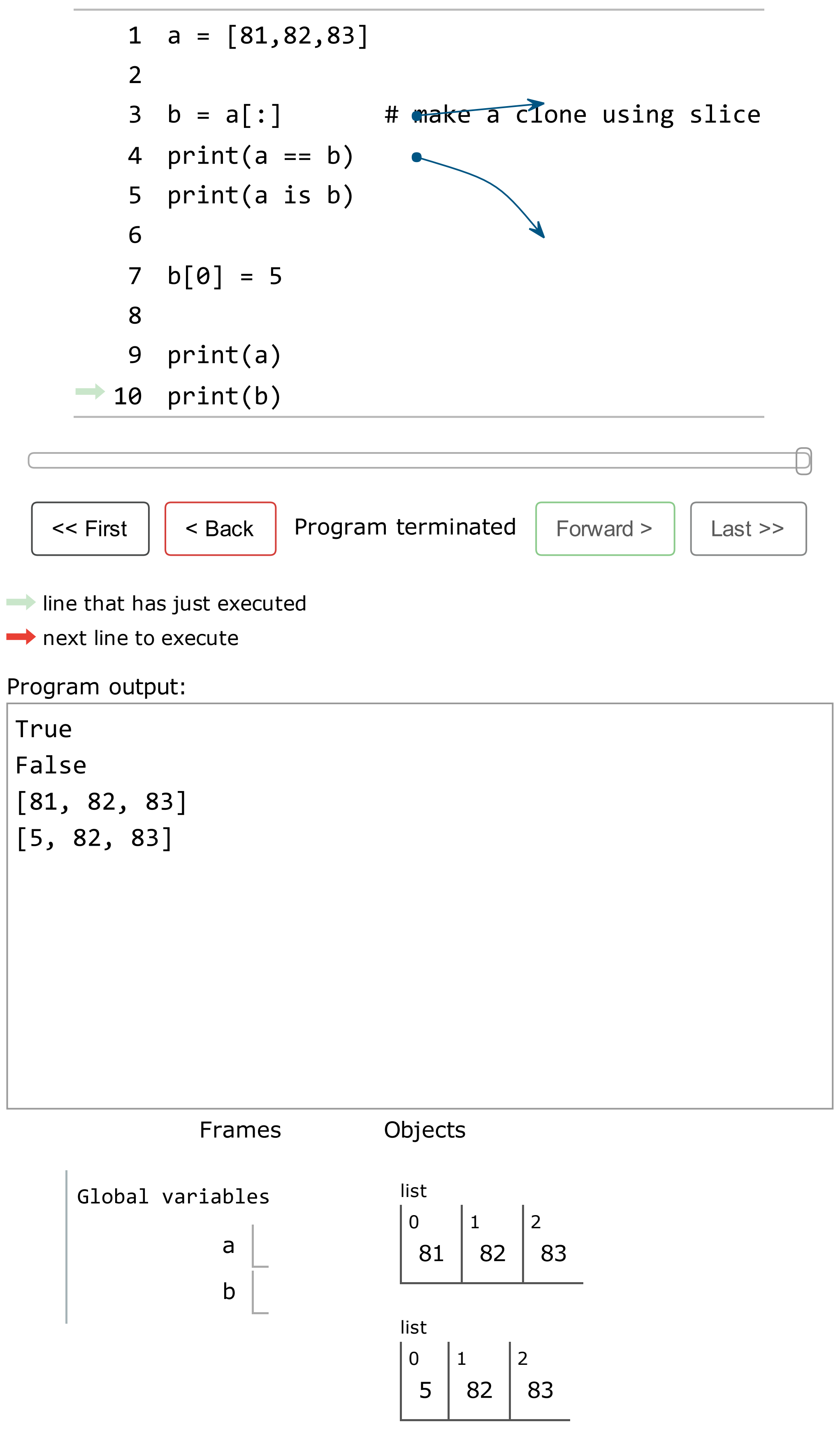
Check your understanding

Cloning Lists

If we want to modify a list and also keep a copy of the original, we need to be able to make a copy of the list itself, not just the reference. This process is sometimes called **cloning**, to avoid the ambiguity of the word copy.

The easiest way to clone a list is to use the slice operator.

Taking any slice of `a` creates a new list. In this case the slice happens to consist of the whole list.



```python
a = [81,82,83]

b = a[:]        # make a clone using slice
print(a == b)
print(a is b)

b[0] = 5

print(a)
print(b)
```

line that has just executed

next line to execute

Program output:

```
True
False
[81, 82, 83]
[5, 82, 83]
```

CodeLens: 4 (chp09_is4)

Now we are free to make changes to `b` without worrying about `a`. Again, we can clearly see in codelens that `a` and `b` are entirely different list objects.

Repetition and References

We have already seen the repetition operator working on strings as well as lists. For example,

```
origlist = [45, 76, 34, 55]
print(origlist*3)

```

ActiveCode: 10 (repref1)

Run

```
[45, 76, 34, 55, 45, 76, 34, 55, 45, 76, 34, 55]
```

With a list, the repetition operator creates copies of the references. Although this may seem simple enough, when we allow a list to refer to another list, a subtle problem can arise.

Consider the following extension on the previous example.

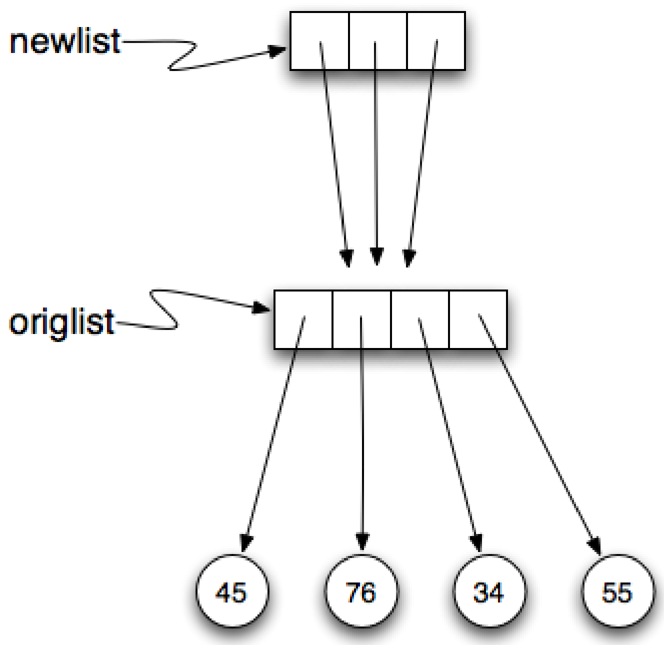
```
origlist = [45, 76, 34, 55]
print(origlist*3)

newlist = [origlist] * 3

print(newlist)

```

ActiveCode: 11 (repref2)

Run

```
[45, 76, 34, 55, 45, 76, 34, 55, 45, 76, 34, 55]
[[45, 76, 34, 55], [45, 76, 34, 55], [45, 76, 34, 55]]
```

`newlist` is a list of three references to `origlist` that were created by the repetition operator. The reference diagram is shown below.



Now, what happens if we modify a value in `origlist`.

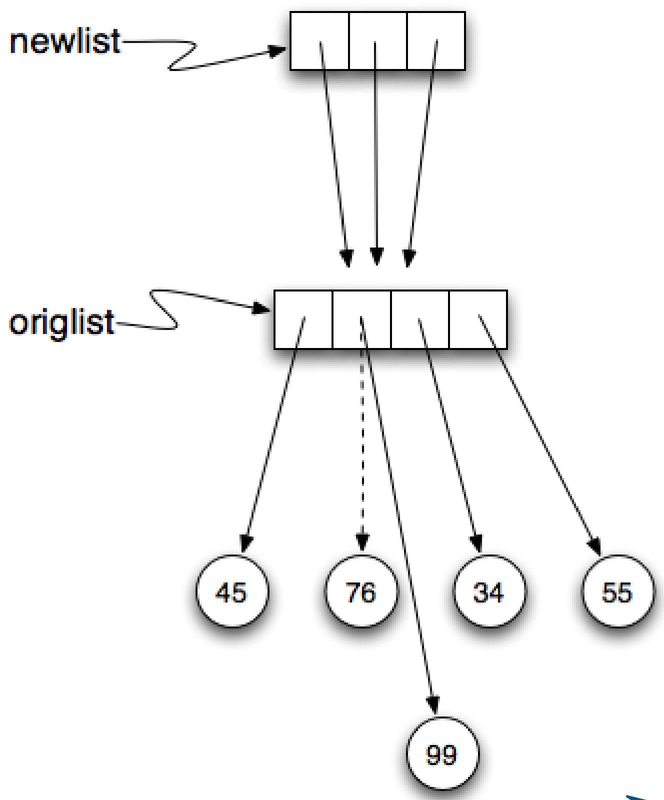
```python
origlist = [45, 76, 34, 55]

newlist = [origlist] * 3

print(newlist)

origlist[1] = 99

print(newlist)

```

ActiveCode: 12 (repref3)

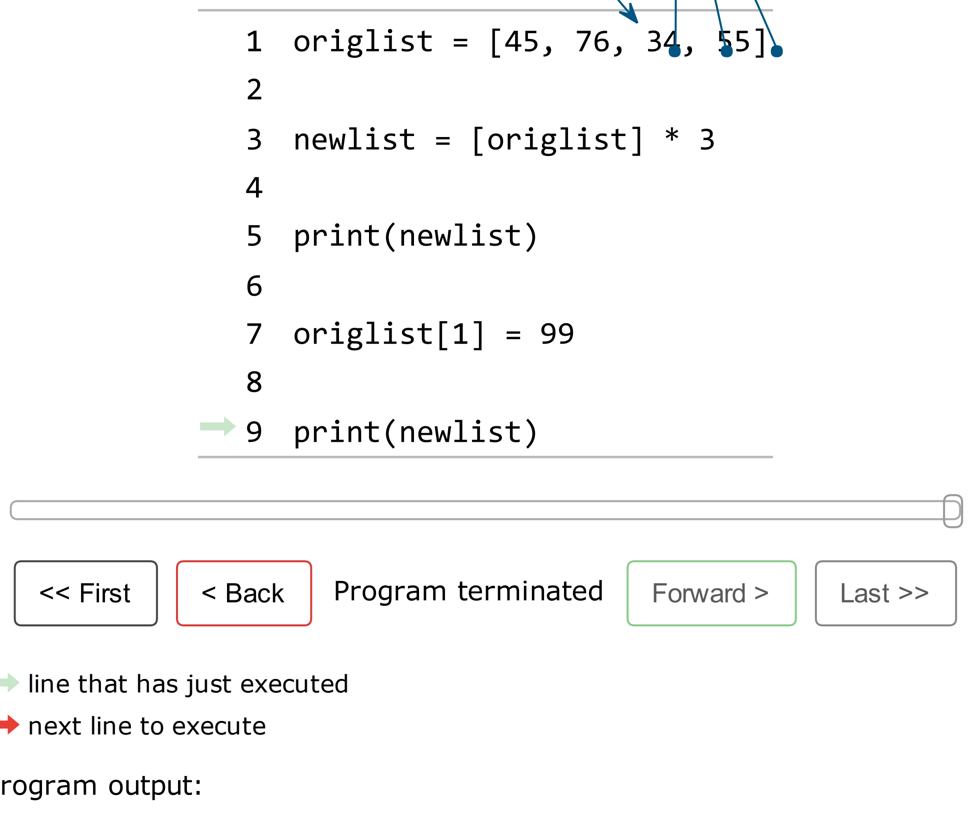
Run

```
[[45, 76, 34, 55], [45, 76, 34, 55], [45, 76, 34, 55]]
[[45, 99, 34, 55], [45, 99, 34, 55], [45, 99, 34, 55]]
```

`newlist` shows the change in three places. This can easily be seen by noting that in the reference diagram, there is only one `origlist`, so any changes to it appear in all three references from `newlist`.



Here is the same example in codelens. Step through the code until paying particular attention to the result of executing the assignment statement `origlist[1] = 99`.



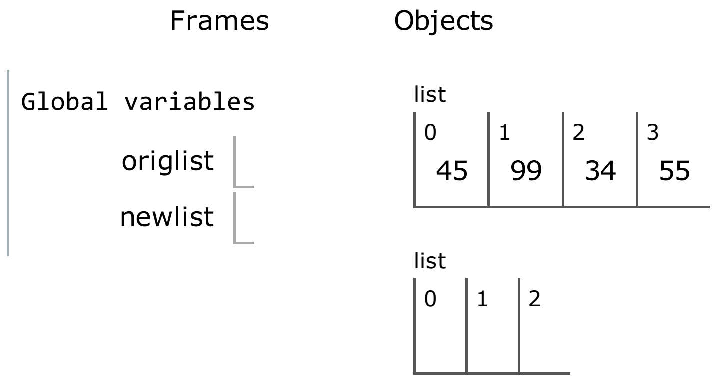
```
origlist = [45, 76, 34, 55]

newlist = [origlist] * 3

print(newlist)

origlist[1] = 99

print(newlist)
```

line that has just executed

next line to execute

Program output:

```
[[45, 76, 34, 55], [45, 76, 34, 55], [45, 76, 34, 55]]
[[45, 99, 34, 55], [45, 99, 34, 55], [45, 99, 34, 55]]
```



CodeLens: 5 (reprefstep)

Check your understanding

List Methods

The dot operator can also be used to access built-in methods of list objects. `append` is a list method which adds the argument passed to it to the end of the list. Continuing with this example, we show several other list methods. Many of them are easy to understand.

```
print(mylist.count(5))

mylist.reverse()
print(mylist)

mylist.sort()
print(mylist)

mylist.remove(5)
print(mylist)

lastitem = mylist.pop()
print(lastitem)
print(mylist)

```

ActiveCode: 13 (chp09_meth1)

Run

```
[5, 27, 3, 12]
[5, 12, 27, 3, 12]
2
3
1
[12, 3, 27, 12, 5]
[3, 5, 12, 12, 27]
[3, 12, 12, 27]
27
[3, 12, 12]
```

There are two ways to use the `pop` method. The first, with no parameter, will remove and return the last item of the list. If you provide a parameter for the position, `pop` will remove and return the item at that position. Either way the list is changed.

The following table provides a summary of the list methods shown above. The column labeled result gives an explanation as to what the return value is as it relates to the new value of the list. The word **mutator** means that the list is changed by the method but nothing is returned (actually `None` is returned). A **hybrid** method is one that not only changes the list but also returns a value as its result. Finally, if the result is simply a return, then the list is unchanged by the method.

Be sure to experiment with these methods to gain a better understanding of what they do.

Method	Parameters	Result	Description
append	item	mutator	Adds a new item to the end of a list

insert	position, item	mutator	Inserts a new item at the position given
pop	none	hybrid	Removes and returns the last item
pop	position	hybrid	Removes and returns the item at position
sort	none	mutator	Modifies a list to be sorted
reverse	none	mutator	Modifies a list to be in reverse order
index	item	return idx	Returns the position of first occurrence of item
count	item	return ct	Returns the number of occurrences of item
remove	item	mutator	Removes the first occurrence of item

Details for these and others can be found in the Python Documentation (http://docs.python.org/py3k/library/stdtypes.html#sequence-types-str-bytes-bytearray-list-tuple-range).

It is important to remember that methods like `append` , `sort` , and `reverse` all return `None` . This means that re-assigning `mylist` to the result of sorting `mylist` will result in losing the entire list. Calls like these will likely never appear as part of an assignment statement (see line 8 below).

```
 1  mylist = []
 2  mylist.append(5)
 3  mylist.append(27)
 4  mylist.append(3)
 5  mylist.append(12)
 6  print(mylist)
 7
 8  mylist = mylist.sort()   #probably an error
 9  print(mylist)
10
```

ActiveCode: 14 (chp09_meth2)

Run

```
[5, 27, 3, 12]
None
```

Check your understanding

Note

This workspace is provided for your convenience. You can use this activecode window to try out anything you like.

```python
mylist = []
mylist.append(5)
mylist.append(27)
mylist.append(3)
mylist.append(12)
print(mylist)

mylist1 = mylist.sort()    #probably an error
print(mylist)

```

ActiveCode: 15 (scratch_09_03)

Run

```
[5, 27, 3, 12]
[3, 5, 12, 27]
```