Lists

A **list** is a sequential collection of Python data values, where each value is identified by an index. The values that make up a list are called its **elements**. Lists are similar to strings, which are ordered collections of characters, except that the elements of a list can have any type and for any one list, the items can be of different types.

List Values

There are several ways to create a new list. The simplest is to enclose the elements in square brackets (`[` and `]`).

```
[10, 20, 30, 40]
["spam", "bungee", "swallow"]
```

The first example is a list of four integers. The second is a list of three strings. As we said above, the elements of a list don't have to be the same type. The following list contains a string, a float, an integer, and another list.

```
["hello", 2.0, 5, [10, 20]]
```

A list within another list is said to be **nested** and the inner list is often called a **sublist**. Finally, there is a special list that contains no elements. It is called the empty list and is denoted `[]` .

As you would expect, we can also assign list values to variables and pass lists as parameters to functions.

```
1  vocabulary = ["iteration", "selection", "control"]
2  numbers = [17, 123]
3  empty = []
4  mixedlist = ["hello", 2.0, 5*2, [10, 20]]
5
6  print(numbers)
7  print(mixedlist)
8  newlist = [ numbers, vocabulary ]
9  print(newlist)
10
```

ActiveCode: 1 (chp09_01)

Run

```
[17, 123]
['hello', 2, 10, [10, 20]]
[[17, 123], ['iteration', 'selection', 'control']]
```

Check your understanding

List Length

As with strings, the function `len` returns the length of a list (the number of items in the list). However, since lists can have items which are themselves lists, it important to note that `len` only returns the top-most length. In other words, sublists are considered to be a single item when counting the length of the list.

```
alist =  ["hello", 2.0, 5, [10, 20]]
print(len(alist))
print(len(['spam!', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]))

```

ActiveCode: 2 (chp09_01a)

Run

```
4
4
```

Check your understanding

Accessing Elements

The syntax for accessing the elements of a list is the same as the syntax for accessing the characters of a string. We use the index operator (`[]` – not to be confused with an empty list). The expression inside the brackets specifies the index. Remember that the indices start at 0. Any integer expression can be used as an index and as with strings, negative index values will locate items from the right instead of from the left.

```
numbers = [17, 123, 87, 34, 66, 8398, 44]
print(numbers[2])
print(numbers[9-8])
print(numbers[-2])
print(numbers[len(numbers)-1])

```

ActiveCode: 3 (chp09_02)

Run

```
87
123
8398
44
```

Check your understanding

List Membership

`in` and `not in` are boolean operators that test membership in a sequence. We used them previously with strings and they also work here.

```
1 fruit = ["apple","orange","banana","cherry"]
2
3 print("apple" in fruit)
4 print("pear" in fruit)
5
```

ActiveCode: 4 (chp09_4)

Run

```
True
False
```

Check your understanding

Concatenation and Repetition

Again, as with strings, the `+` operator concatenates lists. Similarly, the `*` operator repeats the items in a list a given number of times.

```
1 fruit = ["apple","orange","banana","cherry"]
2 print([1,2] + [3,4])
3 print(fruit+[6,7,8,9])
4
5 print([0] * 4)
6 print([1,2,["hello","goodbye"]]*2)
7
```

ActiveCode: 5 (chp09_5)

Run

```
[1, 2, 3, 4]
['apple', 'orange', 'banana', 'cherry', 6, 7, 8, 9]
[0, 0, 0, 0]
[1, 2, ['hello', 'goodbye'], 1, 2, ['hello', 'goodbye']]
```

It is important to see that these operators create new lists from the elements of the operand lists. If you concatenate a list with 2 items and a list with 4 items, you will get a new list with 6 items (not a list with two sublists). Similarly, repetition of a list of 2 items 4 times will give a list with 8 items.

One way for us to make this more clear is to run a part of this example in codelens. As you step thru the code, you will see the variables being created and the lists that they refer to. Pay particular attention to the fact that when `newlist` is created by the statement `newlist = fruit + numlist`, it refers to a completely new list formed by making copies of the items from `fruit` and `numlist`. You can see this very clearly in the codelens object diagram. The objects are different.

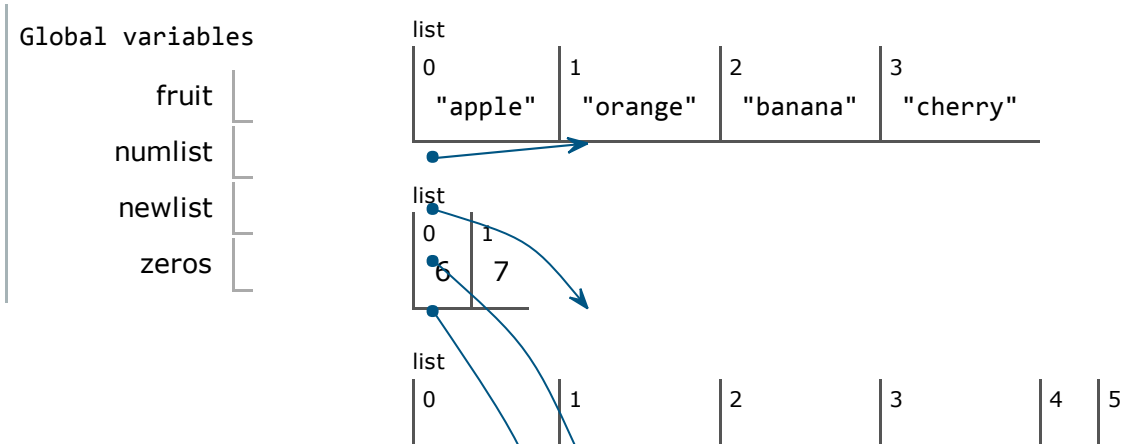
```
1  fruit = ["apple","orange","banana","cherry"]
2  numlist = [6,7]
3
4  newlist = fruit + numlist
5
6  zeros = [0] * 4
```

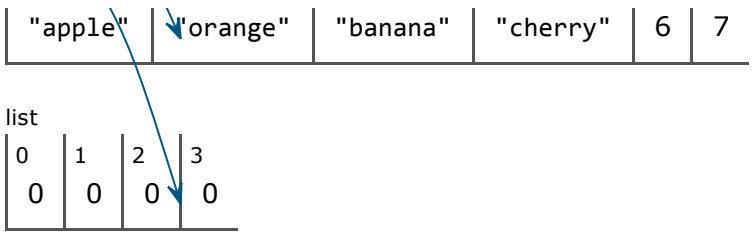
<< First | < Back | Program terminated | Forward > | Last >>

line that has just executed

next line to execute

Frames Objects





CodeLens: 1 (chp09_concatid)

In Python, every object has a unique identification tab. Likewise, there is a built-in function that can be called on any object to return its unique id. The function is appropriately called `id` and takes a single parameter, the object that you are interested in knowing about. You can see in the example below that a real id is usually a very large integer value (corresponding to an address in memory).

```
>>> alist = [4,5,6]
>>> id(alist)
4300840544
>>>
```

Check your understanding

List Slices

The slice operation we saw with strings also work on lists. Remember that the first index is the starting point for the slice and the second number is one index past the end of the slice (up to but not including that element). Recall also that if you omit the first index (before the colon), the slice starts at the beginning of the sequence. If you omit the second index, the slice goes to the end of the sequence.

```
1 a_list = ['a', 'b', 'c', 'd', 'e', 'f']
2 print(a_list[1:3])
3 print(a_list[:4])
4 print(a_list[3:])
5 print(a_list[:])
6
```

ActiveCode: 6 (chp09_6)

Run

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

Check your understanding