The inner life of lists

Now we want to show you one important, and very surprising, feature of lists, which strongly distinguishes them from ordinary variables.

We want you to memorize it - it may affect your future programs, and cause severe problems if forgotten or overlooked.

Take a look at the snippet below.

```
1  list1 = [1]
2  list2 = list1
3  list1[0] = 2
4  print(list2)
```

The program:

- creates a one-element list named `list1`;
- assigns it to a new list named `list2`;
- changes the only element of `list1`;
- prints out `list2`.

The surprising part is the fact that the program will output: `[2]`, not `[1]`, which seems to be the obvious solution.

Lists (and many other complex Python entities) are stored in different ways than ordinary (scalar) variables.

You could say that:

- the name of an ordinary variable is the **name of its content**;
- the name of a list is the name of a **memory location where the list is stored**.

Read these two lines once more - the difference is essential for understanding what we are going to talk about next.

The assignment: `list2 = list1` copies the name of the array, not its contents. In effect, the two names (`list1` and `list2`) identify the same location in the computer memory. Modifying one of them affects the other, and vice versa.

How do you cope with that?

Powerful slices

Fortunately, the solution is at your fingertips - its name is the **slice**.

A slice is an element of Python syntax that allows you to **make a brand new copy of a list, or parts of a list**.

It actually copies the list's contents, not the list's name.

This is exactly what you need. Take a look at the snippet below:

```
list1 = [1]

list2 = list1[:]

list1[0] = 2

print(list2)
```

Its output is `[1]`.

This inconspicuous part of the code described as `[:]` is able to produce a brand new list.

One of the most general forms of the slice looks as follows:

```
myList[start:end]
```

As you can see, it resembles indexing, but the colon inside makes a big difference.

A slice of this form **makes a new (target) list, taking elements from the source list - the elements of the indices from start to** `end - 1`.

Note: not to `end` but to `end - 1`. An element with an index equal to `end` is the first element which **does not take part in the slicing**.

Using negative values for both start and end is possible (just like in indexing).

Take a look at the snippet:

```
myList = [10, 8, 6, 4, 2]

newList = myList[1:3]

print(newList)
```

The `newList` list will have `end - start` (3 - 1 = 2) elements - the ones with indices equal to `1` and `2` (but not `3`).

The snippet's output is: `[8, 6]`

Slices - negative indices

Look at the snippet below:

```
myList[start:end]
```

To repeat:

- `start` is the index of the first element **included in the slice**;

- `end` is the index of the first element **not included in the slice.**

This is how **negative indices** work with the slice:

```
myList = [10, 8, 6, 4, 2]
newList = myList[1:-1]
print(newList)
```

The snippet's output is: `[8, 6, 4]`.

If the `start` specifies an element lying further than the one described by the `end` (from the list's beginning point of view), the slice will be **empty**:

```
myList = [10, 8, 6, 4, 2]
newList = myList[-1:1]
print(newList)
```

The snippet's output is: `[]`.

Slices: continued

If you omit the `start` in your slice, it is assumed that you want to get a slice beginning at the element with index `0`.

In other words, the slice of this form:

```
myList[:end]
```

is a more compact equivalent of:

```
myList[0:end]
```

Look at the snippet below:

```
myList = [10, 8, 6, 4, 2]

newList = myList[:3]

print(newList)
```

This is why its output is: `[10, 8, 6]`.

Similarly, if you omit the `end` in your slice, it is assumed that you want the slice to end at the element with the index `len(myList)`.

In other words, the slice of this form:

```
myList[start:]
```

is a more compact equivalent of:

```
myList[start:len(myList)]
```

Look at the following snippet:

```
myList = [10, 8, 6, 4, 2]

newList = myList[3:]

print(newList)
```

Its output is therefore: `[4, 2]`.

Slices: continued

As we've said before, omitting both `start` and `end` makes a **copy of the whole list**:

```
myList = [10, 8, 6, 4, 2]

newList = myList[:]

print(newList)
```

The snippet's output is: `[10, 8, 6, 4, 2]`.

The previously described `del` instruction is able to **delete more than just a list's element at once - it can delete slices too**:

```
myList = [10, 8, 6, 4, 2]

del myList[1:3]

print(myList)
```

Note: in this case, the slice **doesn't produce any new list**!

The snippet's output is: `[10, 4, 2]`.

Deleting **all the elements** at once is possible too:

```
myList = [10, 8, 6, 4, 2]

del myList[:]

print(myList)
```

The list becomes empty, and the output is: `[]`.

Removing the slice from the code changes its meaning dramatically.

Take a look:

`myList = [10, 8, 6, 4, 2]`

`del myList`

`print(myList)`

The `del` instruction will **delete the list itself, not its content**.

The `print()` function invocation from the last line of the code will then cause a runtime error.

The `in` and `not in` operators

Python offers two very powerful operators, able to **look through the list in order to check whether a specific value is stored inside the list or not**.

These operators are:

`elem in myList`

`elem not in myList`

The first of them (`in`) checks if a given element (its left argument) is currently stored somewhere inside the list (the right argument) - the operator returns `True` in this case.

The second (`not in`) checks if a given element (its left argument) is absent in a list - the operator returns `True` in this case.

Look at the code below.

```
1  myList = [0, 3, 12, 8, 2]
2
3  print(5 in myList)
4  print(5 not in myList)
5  print(12 in myList)
```

The snippet shows both operators in action. Can you guess its output? Run the program to check if you were right.

Lists - some simple programs

Now we want to show you some simple programs utilizing lists.

The first of them tries to find the greater value in the list. Look at the code below.

```
1  myList = [17, 3, 11, 5, 1, 9, 7, 15, 13]
2  largest = myList[0]
3
4  for i in range(1, len(myList)):
5      if myList[i] > largest:
6          largest = myList[i]
7
8  print(largest)
```

The concept is rather simple - we temporarily assume that the first element is the largest one, and check the hypothesis against all the remaining elements in the list.

The code outputs `17` (as expected).

The code may be rewritten to make use of the newly introduced form of the `for` loop:

```
myList = [17, 3, 11, 5, 1, 9, 7, 15, 13]
largest = myList[0]

for i in myList:
      if i > largest:
            largest = i

print(largest)
```

The program above performs one unnecessary comparison, when the first element is compared with itself, but this isn't a problem at all.

The code outputs `17`, too (nothing unusual).

If you need to save computer power, you can use a slice:

```
myList = [17, 3, 11, 5, 1, 9, 7, 15, 13]
largest = myList[0]

for i in myList[1:]:
      if i > largest:
            largest = i
print(largest)
```

The question is: which of these two actions consumes more computer resources - just one comparison, or slicing almost all of a list's elements?

Lists - some simple programs

Now let's find the location of a given element inside a list:

```
myList = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

toFind = 5

found = False

for i in range(len(myList)):

      found = myList[i] == toFind

      if found:

      break
```

```
if found:
    print("Element found at index", i)
else:
    print("absent")
```

Note:

- the target value is stored in the `toFind` variable;
- the current status of the search is stored in the `found` variable (`True`/`False`)
- when `found` becomes `True`, the `for` loop is exited.

Let's assume that you've chosen the following numbers in the lottery: `3`, `7`, `11`, `42`, `34`, `49`.

The numbers that have been drawn are: `5`, `11`, `9`, `42`, `3`, `49`.

The question is: how many numbers have you hit?

The program will give you the answer:

```
drawn = [5, 11, 9, 42, 3, 49]
bets = [3, 7, 11, 42, 34, 49]
hits = 0
for number in bets:
    if number in drawn:
        hits += 1
print(hits)
```

Note:

- the `drawn` list stores all the drawn numbers;
- the `bets` list stores your bets;
- the `hits` variable counts your hits.

The program output is: `4`.

Key takeaways

1. If you have a list `l1`, then the following assignment: `l2 = l1` does not make a copy of the `l1` list, but makes the variables `l1` and `l2` **point to one and the same list in memory**. For example:

```
vehiclesOne = ['car', 'bicycle', 'motor']
print(vehiclesOne) # outputs: ['car', 'bicycle', 'motor']

vehiclesTwo = vehiclesOne
del vehiclesOne[0] # deletes 'car'
print(vehiclesTwo) # outputs: ['bicycle', 'motor']
```

2. If you want to copy a list or part of the list, you can do it by performing **slicing**:

```
colors = ['red', 'green', 'orange']

copyWholeColors = colors[:] # copy the whole list
copyPartColors = colors[0:2] # copy part of the list
```

3. You can use **negative indices** to perform slices, too. For example:

```
sampleList = ["A", "B", "C", "D", "E"]
newList = sampleList[2:-1]
print(newList) # outputs: ['C', 'D']
```

4. The `start` and `end` parameters are **optional** when performing a slice: `list[start:end]`, e.g.:

```
myList = [1, 2, 3, 4, 5]
sliceOne = myList[2: ]
sliceTwo = myList[ :2]
sliceThree = myList[-2: ]

print(sliceOne) # outputs: [3, 4, 5]
print(sliceTwo) # outputs: [1, 2]
print(sliceThree) # outputs: [4, 5]
```

5. You can **delete slices** using the `del` instruction:

```
myList = [1, 2, 3, 4, 5]
del myList[0:2]
print(myList) # outputs: [3, 4, 5]

del myList[:]
print(myList) # deletes the list content, outputs: []
```

6. You can test if some items **exist in a list or not** using the keywords `in` and `not in`, e.g.:

```
myList = ["A", "B", 1, 2]

print("A" in myList) # outputs: True

print("C" not in myList) # outputs: True

print(2 not in myList) # outputs: False
```