Operations that modify Lists in Place

MARCH 21ST, 2012

Lists and strings also have important differences

- In Python some data types are *mutable,* i.e., they can be modified in place.
- Of the data types we have seen so far, e.g., `int`, `long`, `float`, `bool`, `str`, and `list`, only `list` is mutable.

Example:

```
>>> L = [3, 4, 5]
>>> type(L)
<type 'list'>
>>> L[0] = 8
>>> L
[8, 4, 5]
```

By doing an assignment to `L[0]`, we have replaced the first element in the list `L`.

```
>>> s = "hello"
>>> type(s)
<type 'str'>
>>> s[0]
'h'
>>> s[0] = "t"
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
TypeError: 'str' object does not support item assignment
```

We can examine elements in the string s in a similar manner, but we cannot assign anything to `s[0]`

Looking behind the scenes

Example:

```
>>> id(L)
12494888
>>> L[0] = 11
>>> id(L)
12494888
```

The *id* function when applied to a variable name, returns the location pointed to by that variable. Notice how the location of L does not change as a result of replacing the first element by something else.

```
>>> n = 10
>>> id(n)
10022540
>>> n = 12
>>> id(n)
10022516
```

An assignment to an int variable does not modify the variable "in place." The variable ends up pointing to another location.

List operations that modify a list "in place"

Replacing single elements or slices of lists

- L[0] = 10,
- L[3:5] = [10, 12],
- L[3:10:2] = [12,14,16, 18]

Deleting a list or its parts

- del L
- del L[3]
- del L[3:5]
- del L[3:10:2]

More such operations

Try and understand all of these operations.

- L.append("hi")
- L.extend(["good"])
- L.insert(4, "bye")
- L.pop(), L.pop(4)
- L.remove("hello")

None of these work on strings.

And here are the last two:

- L.reverse(), L.sort()

Behind the scenes

- The difference between objects of type `list` and objects of other types is due to an important difference in implementation.
- Consider the assignment: `L = [3, 4, 5]`
- We might think that after this assignment, `L` points to the list `[3, 4, 5]`. But no! `L` points to something that in turn points to `[3, 4, 5]`.
- In programming language terminology, we say L points to a *reference* to `[3, 4, 5]`.

Implications: list assignment

- Consider the example:

```
>>> L =  [3, 4, 5]
>>> LL = L
>>> L.append(6)
>>> LL
[3, 4, 5, 6]
```

- Notice how when modified L, the list LL also changed. This is not true for any of the data types we have seen so far.

- After the assignment `LL = L`, `LL` points to a reference that points to the same list as `L`.

Another example of list assignment

```
>>> L = [3, 4, 5]
>>> LCopy = L
>>> M = [3, 4, 5]
>>> L == LCopy, LCopy == M, M == L
(True, True, True)
>>> L[0] = 9
>>> L == LCopy, LCopy == M, M == L
(True, False, False)
```

Implications: Mutation in Functions

```
def test(L):
    L[0] = 7
    return sum(L)

J = [3,  4, 5]
print test(J)
print J
```

- Consider the above program. When you run this and print J, you will see that J has become [7, 4, 5].
- When J is sent in as argument to test, L is given a copy of J. But, since J is pointing to a reference to a list, L ends up pointing to a copy of the reference, but to the same physical list.
- This provides another way of communicating between a main program and functions (and between functions).