Home Lesson-5.5

Lesson-5.4

Lists

List Methods

`insert`

We have looked at list methods like `append`, `count` and `index` so far. There are some more interesting methods that will come in handy. `insert` can be used to insert an element in a list at a given position:

```
L = [1, 1, 2, 3, 8]
L.insert(4, 5)
print(L)
```

`list.insert(index, object)` inserts the `object` before `index` in the `list`. In the code given above, the element `5` is inserted before the index `4` in the list `L`. Let us try a few more inserts:

```
L = [10, 20, 30]
L.insert(0, 5)          # L becomes [5, 10, 20, 30]
L.insert(2, 15)         # L becomes [5, 10, 15, 20, 30]
L.insert(4, 25)         # L becomes [5, 10, 15, 20, 25, 30]
L.insert(len(L), 35)    # L becomes [5, 10, 15, 20, 25, 30, 35]
L.insert(20, 40)        # L becomes [5, 10, 15, 20, 25, 30, 35, 40]
```

If the index is greater than the length of the current list, then the element gets added to the end. `insert` is most useful when an element needs to be inserted at the beginning of a list. Inserting an element at the end can be done using `append`.

pop

Consider the following code:

```
L = ['a', 'b', 'c', 'd', 'e', 'f']
index = 1
x = L.pop(index)
print(f'The element {x} at index {index} was removed from the list')
print(f'The current list is {L}')
```

`L.pop(index)` removes the element at `index` in `L` and returns it. If no argument is provided to `pop`, `index` defaults to -1. `index` is thus a default argument for the method `pop`. A default value of -1 means that the last element in the list is removed. To see this an action, execute the following code:

```
L = ['a', 'b', 'c', 'd', 'e', 'f']
x = L.pop()
print(f'The current list is {L}')
```

What happens if you enter an `index` that is out of range?

reverse

A list can be reversed in-place using the following method:

```
L = [1, 2, 3, 4, 5]
print('Before:', L, id(L))
L.reverse()
print('After:', L, id(L))
```

It is called in-place because the list before and after have the same `id`, i.e., they correspond to the same object. One must be careful while using methods that perform operations in-place. A common error is to do something like this:

```
L = [1, 2, 3, 4, 5]
L = L.reverse()
print(L)
```

This prints `None`, which is expected as `reverse` doesn't return a list. But sometimes, one may want to hold on to the original copy as well as its reverse. In such cases, we could do the following:

```
1 L = [1, 2, 3, 4, 5]
2 L_reversed = L.copy()
3 L_reversed.reverse()
4 print('Original list:', L)
5 print('Reversed list:', L_reversed)
```

Why did we have to make a copy in line-2?

`sort`

Another useful method is `sort` which is used to sort lists in-place:

```
1 L = [2, 1, 5, 6, 4, 3]
2 print('Before', L)
3 L.sort()
4 print('After', L)
```

Though this appears to be such a simple function to call, sorting is a non-trivial algorithm. We will be studying various algorithms to sort a sequence of items in the next course on data structures and algorithms.

`remove`

Now for some destructive functions:

```
1 L = [1, 2, 3, 4, 5] * 2
2 print('Before', L)
3 L.remove(1)
4 print('After', L)
```

`L.remove(x)` removes the first (leftmost) occurrence of the element `x` in the list `L`. Trying to remove an element that is not there in the list will raise a `ValueError` with the message `list.remove(x): x not in list`. A safe way to remove items is as follows:

```
1 # x is the item to be removed; L is the list
2 if x in L:
3     L.remove(x)
```

How is `remove` different from `pop`?

Stack

A list along with the methods `append` and `pop` simulate a data structure called **stack**. A stack is a storage mechanism where the last item added to it is the first item to be removed. This is analogous to a stack of books. The topmost book in the stack is the most recent addition. When we want to remove books from this stack, the topmost book is the first to be removed. There is a catchy mnemonic for this, LIFO: Last In First Out.

```
# Start with an empty stack
stack = [ ]
# Append items to end of the stack; also called a push operation
stack.append('Harry Potter and the Philosopher\'s Stone')
stack.append('Harry Potter and the Chamber of Secrets')
# State of the stack
print(stack)
# Remove items from the end of the stack; also called a pop operation
stack.pop()
# State of the stack
print(stack)
```

Queue

A list along with the methods `insert` and `pop` simulate a data structure called **queue**. A queue is a storage mechanism where the first item added to it is the first to be removed. This is analogous to any queue that we encounter in real life, say at a billing counter. The first person to stand in the queue, is the first to be served, and naturally the first to exit the queue. The mnemonic for this is FIFO: First in First Out.

```
# Start with an empty queue
queue = [ ]
# Insert elements at the beginning of the queue
queue.insert(0, 'Customer-1')
queue.insert(0, 'Customer-2')
# State of the queue
print(queue)
# Remove items from the queue
queue.pop()
# State of the queue
print(queue)
```

Strings and Lists

`split`

Lists make a frequent appearance while processing strings. Consider the following problem:

> Accept a sentence as input and find the number of words in it. Assume that it is a simple sentence with a single space separating consecutive words. There are no other punctuation marks in the sentence.

Let us look at a "list-less" solution first:

Solution-1

```
sentence = 'this sentence is false' # a simple sentence
count = 1
for char in sentence:
    if char == ' ':
        count += 1
print(count)
```

We just scanned the sentence character by character and checked the number of spaces. The total number of words is one more than the number of spaces. As an aside, the sentence that we are dealing with is an example of a paradoxical statement. It can't be true or false: if it is true then it is false, if it is false then it is true! Back to Python, we shall look at the solution that uses lists.

Solution-2

```
sentence = 'this sentence is false' # a simple sentence
words = sentence.split(' ')         # space is the delimiter used
count = len(words)
print(count)
```

`split` is a string method that splits a string along a delimiter. A delimiter string is one or more characters that specify where to split the string. The output of the `split` operation is a list of strings that are split along the delimiter. If we print the list `words`, we get the following list: `['this', 'sentence', 'is', 'false']`. Let us take another example:

```
comma_words = 'one,two,three,four'
numbers = comma_words.split(',')
print(numbers)
```

We get `['one', 'two', 'three', 'four']` as the output. Note that we have specified `','` as the delimiter. The delimiter is not limited to characters, it can be any string. For example:

```
some_string = 'allISwell'
words = some_string.split('IS')
print(words)
```

The output is: `['all', 'well']`.

`join`

Just as we went from a string to a list, we can also move from a list of strings to a string. Consider the following problem:

> Accept a sequence of words as input and construct a sentence out of it.

We will first look at a solution that doesn't use lists:

Solution-1

```
words = ['this', 'sentence', 'is', 'false']
sentence = ''
for word in words:
    sentence += word + ' '
print(sentence)
```

Though this solution seems correct, it is wrong by one character! Print the last character in the sentence:

```
print(sentence[-1])
```

It is not the letter `e` but a space. We ended up printing an extra space at the end. This might seem trivial, but programming is all about precision. A better solution is given below:

Solution-2

```
words = ['this', 'sentence', 'is', 'false']
sentence = words[0]
for word in words[1 : ]:
    sentence += ' ' + word
print(sentence)
```

This is more accurate. But it seems clumsy as we had to iterate from the second word in the list. The final solution uses a simple method and is quite sophisticated.

Solution-3

```
words = ['this', 'sentence', 'is', 'false']
sentence = ' '.join(words)
print(sentence)
```

Isn't that a thing of beauty! Just as `split` chops a string along a delimiter, `join` stitches together the strings in a list, and the thread it uses is a space in this case. We could also stitch them together using any other string, let us use a comma instead:

```
words = ['one', 'two', 'three']
sentence = ','.join(words)
print(sentence)
```

This output is `one,two,three`. The stitching seems too tight. Let us give it some space:

```
words = ['one', 'two', 'three']
sentence = ', '.join(words)
print(sentence)
```

Notice the space after the comma. The output is `one, two, three`.