Lists

A data structure consisting of a sequence of elements

Alwin Tareen

The Definition of a List

- A list is a sequence of several elements grouped together under a single name.
- The values in a list can be of any data type(int, float, str, bool) and they are called elements or items.
- Instead of writing a program with many variables, such as `nums0`, `nums1`, `nums2`, etc., you can define a single list called `nums`, and access its members using square bracket notation: `nums[0]`, `nums[1]`, `nums[2]`.
- More importantly, you can put other expressions and variables inside the square brackets, such as `nums[i]` and `nums[i+1]`. This allows us to deal with arbitrarily large data sets, with just a small piece of code.

Creating a List

- There are several ways to create a new list. The simplest way is to enclose several elements, separated by commas, between square brackets:

```
fruit = ["strawberry", 58, 3.5]
```

- This creates a list called `fruit` of length `3`.
- Each element of the list gets a number called its **index**.
- The initial element has index `0`, the next element has index `1`, and so on.
- If we wish to access the elements in the list, we must use square bracket notation as follows:

```
listname[indexnumber]
```

The Elements of a List

- Consider the following list:

```
fruit = ["strawberry", 58, 3.5]
```

- In this list, `fruit[0]` is a variable whose value is the string `"strawberry"`.
- Specifically, we have the following:
  - `fruit[0]` $\rightarrow$ `"strawberry"`
  - `fruit[1]` $\rightarrow$ `58`
  - `fruit[2]` $\rightarrow$ `3.5`
- Unlike strings, lists are mutable, because you can change the order of items in a list, or reassign an item in a list.

Assigning Elements in a List

- When the square bracket operator appears on the left side of the assignment statement, it identifies the element of the list that will be assigned.

```
fruit[1] = 65
```

- Like strings, if you attempt to access an element in the list which does not exist, then `Python` will produce an `IndexError`.

```
fruit[98]
```

- Like strings, list indexes can take on negative values, leading to a wraparound effect.

```
fruit[-1]
```

List Operations

There are a number of operations that work the same for lists as they do for strings.

The length of a list: `len()`

This determines the number of items in a list.

```
scores = [15, 361, 652, 19, 23, 39, 48, 75, 239]
print(len(scores))
```

The `in` operator

This tells you if a list contains a particular item.

```
drinks = ["tea", "juice", "soda"]
print("coffee" in drinks)
print("soda" in drinks)
```

List Slicing

- The slice operator also works on lists.

```
listname[firstindex:secondindex]
```

- If you omit the first index, the slice starts at the beginning.
- If you omit the second index, the slice goes to the end.
- If you omit both indexes, the slice produces a clone of the whole list.
- Since lists are mutable, you can use the slice operator on the left side of the assignment statement, meaning it can update several variables at once.

```
lunch = ["soup", "salad", "rice", "beans"]
lunch[1:3] = ["fries", "noodles"]
```

Looping(Traversing) across a List

- If you only need to read the elements in a list, and not perform any updates or changes to them, then you should use the following `for` loop:

```
snacks = ["chips", "cake", "banana"]
for item in snacks:
    print(item)
```

- However, if you need to alter or update the elements in the list, then you need to use another version of the `for` loop, in which you can directly access the indexes:

```
primes = [11, 13, 17, 19, 23]
for i in range(len(primes)):
    primes[i] = primes[i] * 2
```

The + and * Operators

The concatenation operator: +

The + operator adds one list to the end of the other.

```
food = ["chicken", "beef", "fish"]
supplies = ["soap", "detergent", "napkins"]
groceries = food + supplies
```

The multiplication operator: *

This repeats a list a given number of times.

```
result = [5, 8] * 3
```

The following command is particularly useful for quickly creating a list of zeros:

```
dataset = [0] * 20
```

List Functions

- There are many built-in functions that can be used on lists.
- The functions provide a nice shortcut, because they allow you to process a list, without having to write a loop to do the work.
- When calling these functions, the list is placed inside the parentheses, as an argument.

List Functions

The `max()` function

This returns the maximum of the items in the list. It works with data types which are comparable(ints, strings, etc.)

```
scores = [98.5, 86.2, 91.3, 89.1]
highest = max(scores)
```

The `min()` function

This returns the minimum of the items in the list. It works with data types which are comparable(ints, strings, etc.)

```
temps = [18.2, 15.7, 14.3, 12.1]
coldest = min(temps)
```

List Functions

The `sum()` function

This returns the sum of the items in the list. It only works with numerical data types(ints, floats, etc.)

```
prices = [19.9, 5.75, 8.25, 21.5]
total = sum(prices)
```

List Methods

- `Python` provides many built-in methods that operate on lists.
- Note that there is a big difference in how list methods operate, compared to string methods.
- Since strings are immutable, string methods can't alter or change a string.
- However, this is not the case with lists. List methods can make changes and updates to a list, since lists are mutable.
- Recall the syntax of a method call:

```
listname.methodname(arguments)
```

Enlarging a List

append(x)

This method adds a new element `x` to the end of the list.

```
sports = ["basketball", "football"]
sports.append("badminton")
```

extend(x)

This method takes a list `x` as an argument, and generates a new list consisting of the original list, with `x` concatenated on the end.

```
notes = ["do", "ray", "mi"]
melody = ["fa", "so", "la"]
notes.extend(melody)
```

Enlarging a List

`insert(p, x)`

This inserts element `x` at index position `p` of the list.

```
breakfast = ["eggs", "juice", "toast"]
breakfast.insert(1, "bacon")
```

Rearranging a List

sort()

This method arranges the elements of the list from low to high.

```
expenses = [27.5, 8.3, 19.7, 15.1, 24.2]
expenses.sort()
animals = ["camel", "bear", "goat", "dolphin"]
animals.sort()
```

reverse()

This reverses the order of the elements in the list.

```
meals = ["breakfast", "lunch", "dinner"]
meals.reverse()
```

Reducing a List

`remove(x)`

This method removes the first occurrence of element `x` from the list.

```
snacks = ["pizza", "wings", "soda", "chips"]
snacks.remove("soda")
```

`pop(x)`

This method removes the element at index `x`, and returns its value.

```
drinks = ["tea", "coffee", "cookie", "juice"]
pastry = drinks.pop(2)
```

Reducing a List

The following behaves in a similar manner to `pop(x)`.

`del listname[x]`

This removes the element at index `x`, but it doesn't return the removed value.

```
vegetables = ["celery", "watermelon", "broccoli"]
del vegetables[1]
```

`del listname[firstindex:lastindex]`

This allows you to remove several elements at once, using slicing.

```
chemicals = ["Na", "He", "Si", "Ca", "Au", "Pb"]
del chemicals[1:4]
```

Information about a List's Elements

`count(x)`

This method returns the number of times that the element `x` occurs in the list.

```
moves = ["up", "up", "down", "down", "up", "up"]
moves.count("up")
```

`index(x)`

This method returns the index location of the first occurrence of element `x`.

```
flavour = ["cheese", "bbq", "salty", "bbq"]
flavour.index("bbq")
```

Lists and Strings

Since a string is a sequence of characters, and a list is a sequence of elements, it makes sense that we can convert from one form to the other.

`list(x)`

This is a string function. It takes in a string `x` as an argument, and returns a list of single characters.

```
lunch = "pizza"
letters = list(lunch) # ["p", "i", "z", "z", "a"]
```

Lists and Strings

split()

This is a string method. It allows you to break a string into individual words, as long as those words are separated by spaces.

```
meal = "I like hamburgers"
words = meal.split()
```

split(x)

The parameter `x` represents a single text character. It allows you to specify the delimiter used to divide up the string. It is usually a comma.

```
prices = "93,82,47,49,32,87,54"
stock = prices.split(",")
```

Lists and Strings

`join(x)`

- This is a string method. It takes in a list of strings `x` as an argument, and concatenates all of those string elements together.
- `join(x)` must be called on a delimiter string character, which gets placed between each of the string elements in the result.

```
roster = ["alice", "bob", "carl", "dan"]
delimiter = ","
students = delimiter.join(roster)
```

Copying a List(Cloning)

- In order to make a copy of a list, you must **clone** it in the following manner:

```
menu = ["rice", "noodles", "beef"]
specials = menu[:]
```

The `range()` Function

`range(n)`

This function generates a list of integers from `0` to `n-1`.

- `range(5)` $\rightarrow$ `[0, 1, 2, 3, 4]`

`range(m, n)`

This function generates a list of integers from `m` to `n-1`.

- `range(2, 7)` $\rightarrow$ `[2, 3, 4, 5, 6]`

The `range()` Function

`range(m, n, step)`

This function generates a list of integers from `m` to `n-1`, but those integers are separated by gaps of size `step`.

- `range(2, 15, 3)` $\rightarrow$ `[2, 5, 8, 11, 14]`

If we want the list of integers to proceed backwards, then we can specify a negative value for `step`.

- `range(9, 2, -1)` $\rightarrow$ `[9, 8, 7, 6, 5, 4, 3]`

Updating Elements while Looping

If we want to change or alter the elements in the list as we proceed through the loop, then we need consider the index of each element, `i`.

```
nums = [26, 87, 46, 54, 16, 28, 91, 41, 87, 26]
for i in range(len(nums)):
    nums[i] = nums[i] * 2
```

Note that we cannot make changes to the elements when using a regular `for` loop.

Lists: End of Notes