10-lists-I

February 15, 2022

0.1 Lists I

Until now, you have only used numbers and booleans as data to compute with in this course. Numbers are indeed very useful, but sometimes we need to have data in a more structured way. Data structures serve this purpose. Today we look at one of the simplest, and yet most pervasive, data structures out there: **lists**.

Lists are... well... lists.

Lists can have anything inside: `int`, `float`, `boolean`, and other lists! This is how we write a list in Python:

[1]:
```
# List of ints
[3,2,4,7,2]

# Lists of floats
[1.2, 4.6, 3.0]

# Lists of booleans
[True, False, False]

# Lists of lists of ints
[ [11,12,13],
  [21,22,23],
  [31,32,33] ]
```

[1]:
```
[[11, 12, 13], [21, 22, 23], [31, 32, 33]]
```

Lists can be heterogeneous as well (*this is a python particularity*)

[2]:
```
[True, 4, 3.2, [1,4,5]]
```

[2]:
```
[True, 4, 3.2, [1, 4, 5]]
```

0.1.1 Accessing elements and chunks

Elements in a list an be found by their *index*, the position they are in the list. By convention, indices start on the left and the first index is 0. We can get an element at position `i` by using brackets:

[3]:
```
L = [4,5,3,6,2]
L[2]
```

[3]:
```
3
```

Python allows us to access elements indexing from the back of the list with negative numbers. So the last element is at index -1, the second last at index -2, and so on and so forth.

[4]:
```
L = [4,5,3,6,2]
L[-1]
```

[4]:
```
2
```

It is possible to access sublists of a list, by indexing with two values, a start and end index, separated by a colon. If `L` is a list, `L[i:j]` results in the sublist starting at position `i` and ending at position `j-1` (just like `range`).

This is called *slicing*.

[5]:
```
L = [0,1,2,3,4,5,6]
L[2:5]
```

[5]:
```
[2, 3, 4]
```

Speaking of `range`, we can also get a sublist by skipping some elements by `k` steps.

[6]:
```
L = [0,1,2,3,4,5,6,7,8,9]
L[1:8:2]
```

[6]:
```
[1, 3, 5, 7]
```

Slicing can be abbreviated if: - We are starting from 0: `L[:4]` is the same as `L[0:4]` - We are going until the end: `L[1:]` is the list `L` without the first element.

Slicing with steps can also be abbreviated: - `L[3::2]` starting from element at index 3 until the end, step by 2 - `L[:10:3]` starting from 0 until the tenth element, take every third - `L[::2]` take very second element

0.1.2 List length

The length of a list can be obtained using the function `len` on the list.

[7]:
```
L = [0,1,2,3,4,5,6,7,8,9,10]
n = len(L)
n
```

[7]:
```
11
```

0.1.3 Finding elements

Can you write a function `def find(L, e)` that returns `True` if element `e` is in the list `L`, and returns `False` otherwise?

[8]:
```
def find(L, e):
    # Complete me
    return True
```

Of course you can! But this is such a common operations, that python has it built-in for you.

Checking whether an element is inside a list can be done using the `in` or `not in` operations, which return `True` or `False` (meaning they can be used as conditions for `if`, `elif`, or `while` loops).

[9]:
```
L = [4,2,5,6,1,7,8]
5 in L
```

[9]: `True`

[10]:
```
5 not in L
```

[10]: `False`

0.1.4 Modifying a list

We have seen ways to access a list, now it is time to modify it.

Particular positions of a list can be modified by using a simple assignment, but having the list with the proper index on the left side of `=`

[11]:
```
L = [1,5,1,1,0]
L[4] = 2
L
```

[11]: `[1, 5, 1, 1, 2]`

0.1.5 Adding elements to a list

Lists can be concatenated with each other via the `+` operator:

[12]:
```
L1 = [1,2]
L2 = [3,4]
L1 + L2
```

[12]: `[1, 2, 3, 4]`

In this case, adding one element `e` to a list, at the end or the beginning, can be done by creating a list with only `e` inside, and concatenating this list:

[13]:
```
L = [1,2,3]
L + [0]
```

[13]:
```
[1, 2, 3, 0]
```

[14]:
```
[0] + L
```

[14]:
```
[0, 1, 2, 3]
```

Using slicing, we can add elements at arbitrary positions of a list:

[15]:
```
L = [6,6,6,6]
L[:2] + [0] + L[2:]
```

[15]:
```
[6, 6, 0, 6, 6]
```

0.1.6 Looping through lists

Most of the problems involving lists will requires some sort of loop. Most likely, you will need to loop through the elements of the list to do something with each one of them.

The main way of looping through a list is by taking its length, and building a range with that:

[16]:
```
L = [10,20,30,40,50]
n = len(L)

for i in range(n):
    # Prints the values at each iteration
    print("i =", i, " L[i] =", L[i])

    # Takes the element at position i
    e = L[i]
```

```
i = 0  L[i] = 10
i = 1  L[i] = 20
i = 2  L[i] = 30
i = 3  L[i] = 40
i = 4  L[i] = 50
```

If you decide that you do not need the indices inside the loop, but only the elements of the list, a more consise loop is:

[17]:
```
L = [10,20,30,40,50]

for e in L:
    print("e =", e)
    # You only have access to the element, not the index
```

```
e = 10
e = 20
e = 30
e = 40
e = 50
```

We **strongly** suggest that you stick with the first option in the beginning.

0.2 Tuples

Tuples behave exactly like lists, except that they have fixed size and can never be modified (that means, we cannot modify one element at a specific position). You have used tuples in the course without noticing: when you had to return two values for a function, we asked you to return it as `(value1, value2)`. Well, you returned a tuple!

[18]:
```
T = (1,2,3,4)
T[1]
```

[18]: 2

[19]:
```
T[1:3]
```

[19]: (2, 3)

[20]:
```
T[1:]
```

[20]: (2, 3, 4)

[21]:
```
T = T + (5,6)
T
```

[21]: (1, 2, 3, 4, 5, 6)

0.3 Exercise 0

Implement the function `times10(L)` that returns a list containing the same elements as `L`, but multiplied by 10.

For example, `times10([1,2,3])` should return `[12,20,30]`.

[22]:
```
def times10(L):
    return []
```

0.4 Exercise 1

Find the maximum element of a list.

[23]:
```
def max(L):
    return 42
```

0.5 Exercise 2

You receive a list with zeros and ones corresponding to a binary number. Write the function `binToDec(B)` that returns the decimal representation of `B`.

For example, `binToDec([1,1,0])` should return 6.

[24]:
```
def binToDec(B):
    return 42
```

0.6 Exercise 3

Given a list `L` and element `e`, write the function `count(L, e)` that counts the number of times `e` occurs in `L`.

[25]:
```
def count(L, e):
    return 42
```

0.7 Exercise 4

Implement the function `allTheSame(L)` that returns `True` if all elements of list `L` are the same, and `False` otherwise.

[26]:
```
def allTheSame(L):
    return True
```