Lists – collections of data

Why do we need lists?

It may happen that you have to read, store, process, and finally, print dozens, maybe hundreds, perhaps even thousands of numbers. What then? Do you need to create a separate variable for each value? Will you have to spend long hours writing statements like the one below?

```
var1 = int(input())
var2 = int(input())
var3 = int(input())
var4 = int(input())
var5 = int(input())
var6 = int(input())
:
:
```

If you don't think that this is a complicated task, then take a piece of paper and write a program that:

- reads five numbers,
- prints them in order from the smallest to the largest (NB, this kind of processing is called **sorting**).

You should find that you don't even have enough paper to complete the task.

So far, you've learned how to declare variables that are able to store exactly one given value at a time. Such variables are sometimes called **scalars** by analogy with mathematics. All the variables you've used so far are actually scalars.

Think of how convenient it would be to declare a variable that could **store more than one value**. For example, a hundred, or a thousand or even ten thousand. It would still be one and the same variable, but very wide and capacious. Sounds appealing? Perhaps, but how would it handle such a container full of different values? How would it choose just the one you need?

What if you could just number them? And then say: *give me the value number 2; assign the value number 15; increase the value number 10000*.

We'll show you how to declare such **multi-value variables**. We'll do this with the example we just suggested. We'll write a **program that sorts a sequence of numbers**. We won't be particularly ambitious - we'll assume that there are exactly five numbers.

Let's create a variable called `numbers`; it's assigned with not just one number, but is filled with a list consisting of five values (note: the **list starts with an open square bracket and ends with a closed square bracket**; the space between the brackets is filled with five numbers separated by commas).

```
numbers = [10, 5, 7, 2, 1]
```

Let's say the same thing using adequate terminology: `numbers` **is a list consisting of five values, all of them numbers**. We can also say that this statement creates a list of length equal to five (as in there are five elements inside it).

The elements inside a list **may have different types**. Some of them may be integers, others floats, and yet others may be lists.

Python has adopted a convention stating that the elements in a list are **always numbered starting from zero**. This means that the item stored at the beginning of the list will have the number zero. Since there are five elements in our list, the last of them is assigned the number four. Don't forget this.

You'll soon get used to it, and it'll become second nature.

Before we go any further in our discussion, we have to state the following: our **list is a collection of elements, but each element is a scalar**.

Indexing lists

How do you change the value of a chosen element in the list?

Let's **assign a new value of** `111` **to the first element** in the list. We do it this way:

```
numbers = [10, 5, 7, 2, 1]

print("Original list content:", numbers)

# printing original list content

numbers[0] = 111

print("New list content: ", numbers) # current list content
```

And now we want **the value of the fifth element to be copied to the second element** - can you guess how to do it?

```
numbers = [10, 5, 7, 2, 1]

print("Original list content:", numbers)

# printing original list content

numbers[0] = 111

print("\nPrevious list content:", numbers) # printing previous list
content

numbers[1] = numbers[4] # copying value of the fifth element to the
second

print("New list content:", numbers) # printing current list content
```

The value inside the brackets which selects one element of the list is called an **index**, while the operation of selecting an element from the list is known as **indexing**.

We're going to use the `print()` function to print the list content each time we make the changes. This will help us follow each step more carefully and see what's going on after a particular list modification.

Note: all the indices used so far are literals. Their values are fixed at runtime, but **any expression can be the index**, too. This opens up lots of possibilities.

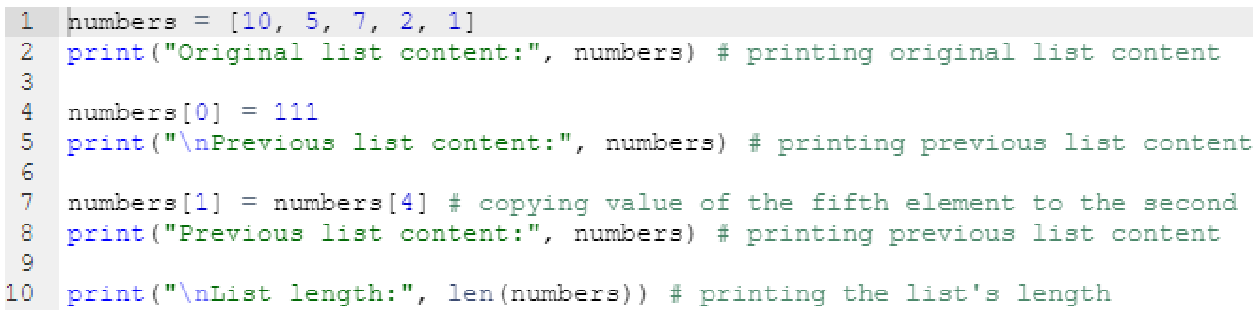
Accessing list content

Each of the list's elements may be accessed separately. For example, it can be printed:

```
print(numbers[0]) # accessing the list's first element
```

Assuming that all of the previous operations have been completed successfully, the snippet will send `111` to the console.

As you can see in the code



```
numbers = [10, 5, 7, 2, 1]
print("Original list content:", numbers) # printing original list content

numbers[0] = 111
print("\nPrevious list content:", numbers) # printing previous list content

numbers[1] = numbers[4] # copying value of the fifth element to the second
print("Previous list content:", numbers) # printing previous list content

print("\nList length:", len(numbers)) # printing the list's length
```

the list may also be printed as a whole - just like here:

```
print(numbers) # printing the whole list
```

As you've probably noticed before, Python decorates the output in a way that suggests that all the presented values form a list. The output from the example snippet above looks like this:

```
[111, 1, 7, 2, 1]
```

The `len()` function

The **length of a list** may vary during execution. New elements may be added to the list, while others may be removed from it. This means that the list is a very dynamic entity.

If you want to check the list's current length, you can use a function named `len()` (its name comes from *length*).

The function takes the **list's name as an argument, and returns the number of elements currently stored** inside the list (in other words - the list's length).

Look at the last line of code in the editor, run the program and check what value it will print to the console. Can you guess?

Removing elements from a list

Any of the list's elements may be **removed** at any time - this is done with an instruction named `del` (delete). Note: it's an **instruction**, not a function.

You have to point to the element to be removed - it'll vanish from the list, and the list's length will be reduced by one.

Look at the snippet below. Can you guess what output it will produce? Run the program in the editor and check.

```
del numbers[1]
```

```
print(len(numbers))
```

```
print(numbers)
```

You can't access an element which doesn't exist - you can neither get its value nor assign it a value. Both of these instructions will cause runtime errors now:

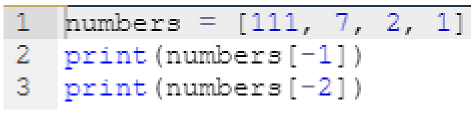
```
print(numbers[4])
```

```
numbers[4] = 1
```

Add the snippet above after the last line of code in the editor, run the program and check what happens.

Note: we've removed one of the list's elements - there are only four elements in the list now. This means that element number four doesn't exist.

Negative indices are legal



```
numbers = [111, 7, 2, 1]
print(numbers[-1])
print(numbers[-2])
```

It may look strange, but negative indices are legal, and can be very useful.

An element with an index equal to `-1` is **the last one in the list**.

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

The example snippet will output `1`. Run the program and check.

Similarly, the element with an index equal to `-2` is **the one before last in the list**.

```
print(numbers[-2])
```

The example snippet will output `2`.

The last accessible element in our list is `numbers[-4]` (the first one) - don't try to go any further!

Functions vs. methods

A **method is a specific kind of function** - it behaves like a function and looks like a function, but differs in the way in which it acts, and in its invocation style.

A **function doesn't belong to any data -** it gets data, it may create new data and it (generally) produces a result.

A method does all these things, but is also able to **change the state of a selected entity**.

A method is owned by the data it works for, while a function is owned by the whole code.

This also means that invoking a method requires some specification of the data from which the method is invoked.

It may sound puzzling here, but we'll deal with it in depth when we delve into object-oriented programming.

In general, a typical function invocation may look like this:

```
result = function(arg)
```

The function takes an argument, does something, and returns a result.

A typical method invocation usually looks like this:

```
result = data.method(arg)
```

Note: the name of the method is preceded by the name of the data which owns the method. Next, you add a **dot**, followed by the **method name**, and a pair of **parenthesis enclosing the arguments**.

The method will behave like a function, but can do something more **-** it can **change the internal state of the data** from which it has been invoked.

You may ask: why are we talking about methods, not about lists?

This is an essential issue right now, as we're going to show you how to add new elements to an existing list. This can be done with methods owned by all the lists, not by functions.

Adding elements to a list: `append()` and `insert()`

A new element may be *glued* to the end of the existing list:

```
list.append(value)
```

Such an operation is performed by a method named `append()`. It takes its argument's value and puts it **at the end of the list** which owns the method.

The list's length then increases by one.

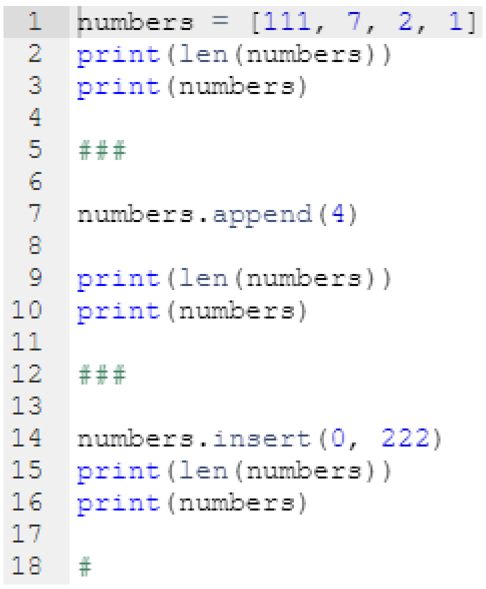
The `insert()` method is a bit smarter - it can add a new element **at any place in the list**, not only at the end.

```
list.insert(location, value)
```

It takes two arguments:

- the first shows the required location of the element to be inserted; note: all the existing elements that occupy locations to the right of the new element (including the one at the indicated position) are shifted to the right, in order to make space for the new element;
- the second is the element to be inserted.

Look at the code below.



```
numbers = [111, 7, 2, 1]
print(len(numbers))
print(numbers)

###

numbers.append(4)

print(len(numbers))
print(numbers)

###

numbers.insert(0, 222)
print(len(numbers))
print(numbers)

#
```

See how we use the `append()` and `insert()` methods. Pay attention to what happens after using `insert()`: the former first element is now the second, the second the third, and so on.

Add the following snippet after the last line of code in the editor:
`numbers.insert(1, 333)`

Print the final list content to the screen and see what happens. The snippet above snippet inserts `333` into the list, making it the second element. The former second element becomes the third, the third the fourth, and so on.

Adding elements to a list: continued

You can **start a list's life by making it empty** (this is done with an empty pair of square brackets) and then adding new elements to it as needed.

Take a look at the snippet in the editor. Try to guess its output after the `for` loop execution. Run the program to check if you were right.

It'll be a sequence of consecutive integer numbers from `1` (you then add one to all the appended values) to `5`.

We've modified the snippet a bit:

```
myList = [] # creating an empty list
for i in range(5):
    myList.insert(0, i + 1)
print(myList)
```

what will happen now? Run the program and check if this time you were right, too.

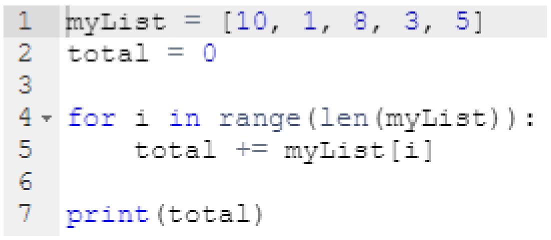
You should get the same sequence, but in **reverse order** (this is the merit of using the `insert()` method).

Making use of lists

The `for` loop has a very special variant that can **process lists** very effectively - let's take a look at that.

Let's assume that you want to **calculate the sum of all the values stored in the `myList` list**.

You need a variable whose sum will be stored and initially assigned a value of `0` - its name will be `total`. (Note: we're not going to name it `sum` as Python uses the same name for one of its built-in functions - `sum()`. Using the same name would generally be considered a bad practice.) Then you add to it all the elements of the list using the `for` loop. Take a look at the snippet below.



```
myList = [10, 1, 8, 3, 5]
total = 0

for i in range(len(myList)):
    total += myList[i]

print(total)
```

Let's comment on this example:

- the list is assigned a sequence of five integer values;
- the `i` variable takes the values `0`, `1`, `2`, `3`, and `4`, and then it indexes the list, selecting the subsequent elements: the first, second, third, fourth and fifth;
- each of these elements is added together by the `+=` operator to the `total` variable, giving the final result at the end of the loop;
- note the way in which the `len()` function has been employed - it makes the code independent of any possible changes in the list's content.

The second face of the `for` loop

But the `for` loop can do much more. It can hide all the actions connected to the list's indexing, and deliver all the list's elements in a handy way.

This modified snippet shows how it works:

```
myList = [10, 1, 8, 3, 5]

total = 0

for i in myList:
    total += i
print(total)
```

What happens here?

- the `for` instruction specifies the variable used to browse the list (`i` here) followed by the `in` keyword and the name of the list being processed (`myList` here)
- the `i` variable is assigned the values of all the subsequent list's elements, and the process occurs as many times as there are elements in the list;
- this means that you use the `i` variable as a copy of the elements' values, and you don't need to use indices;
- the `len()` function is not needed here, either.

Lists in action

Let's leave lists aside for a short moment and look at one intriguing issue.

Imagine that you need to rearrange the elements of a list, i.e., reverse the order of the elements: the first and the fifth as well as the second and fourth elements will be swapped. The third one will remain untouched.

Question: how can you swap the values of two variables?

Take a look at the snippet:

```
variable1 = 1
variable2 = 2
variable2 = variable1
variable1 = variable2
```

If you do something like this, you would **lose the value previously stored** in `variable2`. Changing the order of the assignments will not help. You need a **third variable that serves as an auxiliary storage**.

This is how you can do it:

```
variable1 = 1
variable2 = 2
auxiliary = variable1
variable1 = variable2
variable2 = auxiliary
```

Python offers a more convenient way of doing the swap - take a look:

```
variable1 = 1
variable2 = 2
variable1, variable2 = variable2, variable1
```

Clear, effective and elegant - isn't it?

Lists in action

Now you can easily **swap** the list's elements to **reverse their order**:

```
myList = [10, 1, 8, 3, 5]

myList[0], myList[4] = myList[4], myList[0]

myList[1], myList[3] = myList[3], myList[1]

print(myList)
```

Run the snippet. Its output should look like this:

```
[5, 3, 8, 1, 10]
```

It looks fine with five elements.

Will it still be acceptable with a list containing 100 elements? No, it won't.

Can you use the `for` loop to do the same thing automatically, irrespective of the list's length? Yes, you can.

This is how we've done it:

```
myList = [10, 1, 8, 3, 5]

length = len(myList)

for i in range(length // 2):

    myList[i], myList[length - i - 1] = myList[length - i - 1],
myList[i]

print(myList)
```

Note:

- we've assigned the `length` variable with the current list's length (this makes our code a bit clearer and shorter)
- we've launched the `for` loop to run through its body `length // 2` times (this works well for lists with both even and odd lengths, because when the list contains an odd number of elements, the middle one remains untouched)
- we've swapped the ith element (from the beginning of the list) with the one with an index equal to `(length - i - 1)` (from the end of the list); in our example, for `i` equal to `0` the `(l - i - 1)` gives `4`; for `i` equal to `1`, it gives `3` - this is exactly what we needed.

Lists are extremely useful, and you'll encounter them very often.

Key takeaways

1. The **list is a type of data** in Python used to **store multiple objects**. It is an **ordered and mutable collection** of comma-separated items between square brackets, e.g.:

```
myList = [1, None, True, "I am a string", 256, 0]
```

2. Lists can be **indexed and updated**, e.g.:

```
myList = [1, None, True, 'I am a string', 256, 0]
print(myList[3]) # outputs: I am a string
print(myList[-1]) # outputs: 0

myList[1] = '?'
print(myList) # outputs: [1, '?', True, 'I am a string', 256, 0]

myList.insert(0, "first")
myList.append("last")
print(myList) # outputs: ['first', 1, '?', True, 'I am a string', 256, 0,
'last']
```

3. Lists can be **nested**, e.g.: `myList = [1, 'a', ["list", 64, [0, 1], False]]`.

You will learn more about nesting in module 3.1.7 - for the time being, we just want you to be aware that something like this is possible, too.

4. List elements and lists can be **deleted**, e.g.:

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

del myList # deletes the whole list
```

Again, you will learn more about this in module 3.1.6 - don't worry. For the time being just try to experiment with the above code and check how changing it affects the output.

5. Lists can be **iterated** through using the `for` loop, e.g.:

```
myList = ["white", "purple", "blue", "yellow", "green"]

for color in myList:
    print(color)
```

6. The `len()` function may be used to **check the list's length**, e.g.:

```
myList = ["white", "purple", "blue", "yellow", "green"]
print(len(myList)) # outputs 5

del myList[2]
print(len(myList)) # outputs 4
```

7. A typical **function** invocation looks as follows: `result = function(arg)`, while a typical **method** invocation looks like this:`result = data.method(arg)`.