Python’s List

Built-in Collection

by

Lilian Blot

We have seen that a variable can store a single value\object, so managing hundreds of values means using hundreds of variables.

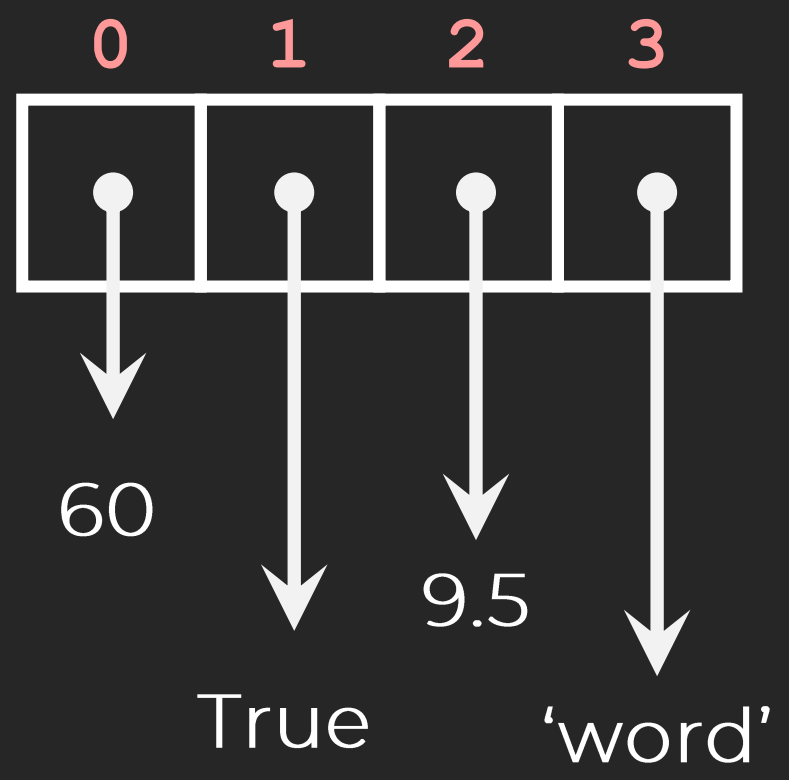
This is not practical; it would be more convenient to store them into a container in order to retrieve and manipulate them easily.

Python provides 4 built-in containers to store multiple values\objects.

1. List
2. Tuple
3. Set
4. Dictionary

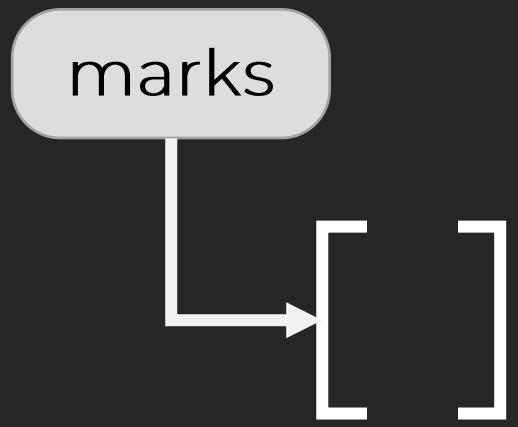
In this video, we will focus on the list data structure.

- A list can contain many different type of object at the same time.
- You can add and remove element of a list.
- A list can have duplicates.
- A list is mutable.



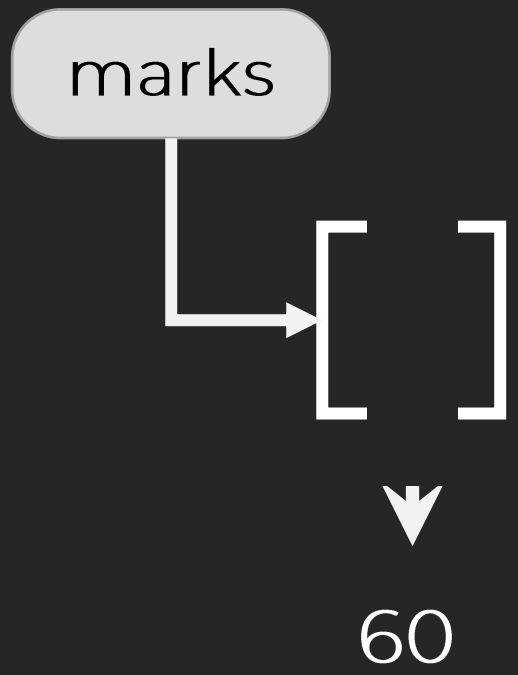
Python shell

```
>>> marks = []
>>>
```



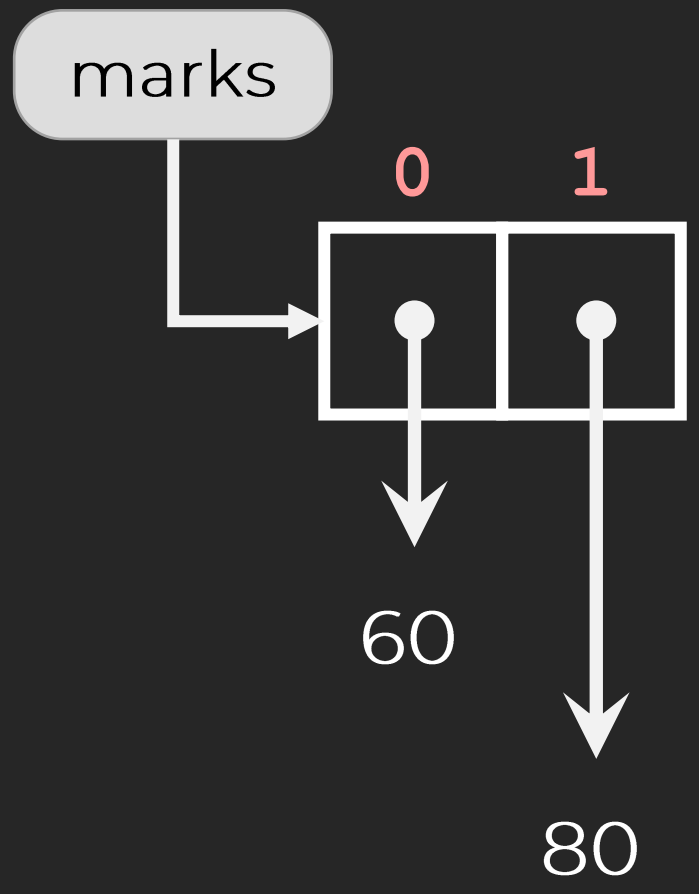
Python shell

```
>>> marks = []
>>> marks.append(60)
>>>
```



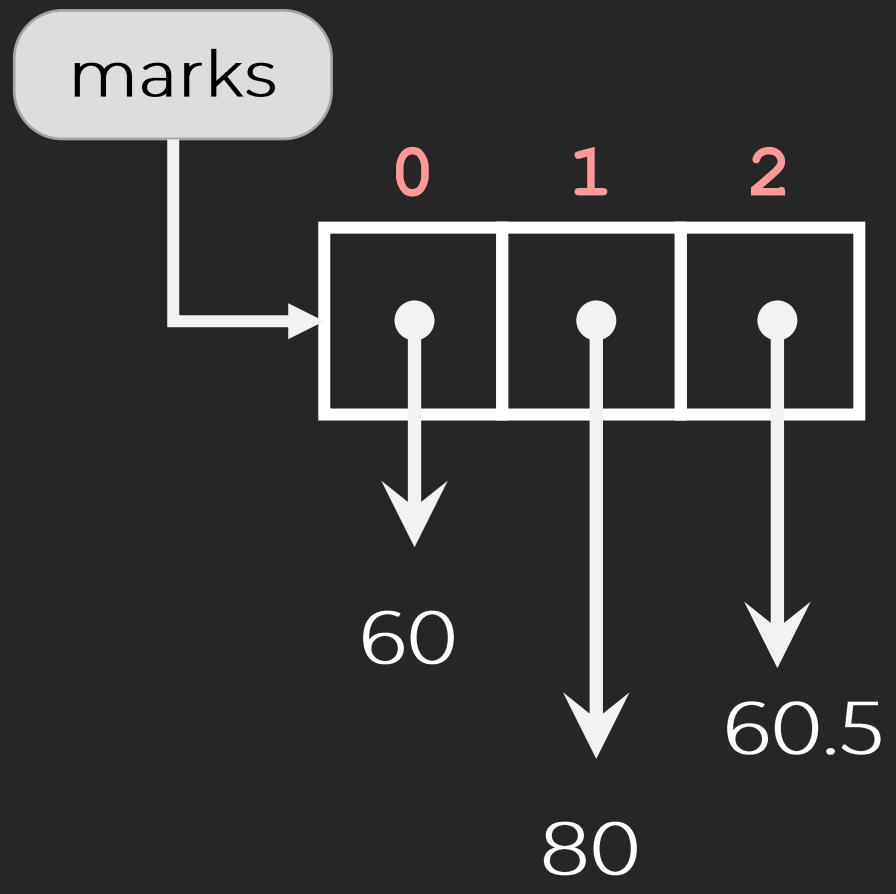
Python shell

```
>>> marks = []
>>> marks.append(60)
>>> marks.append(80)
>>>
```



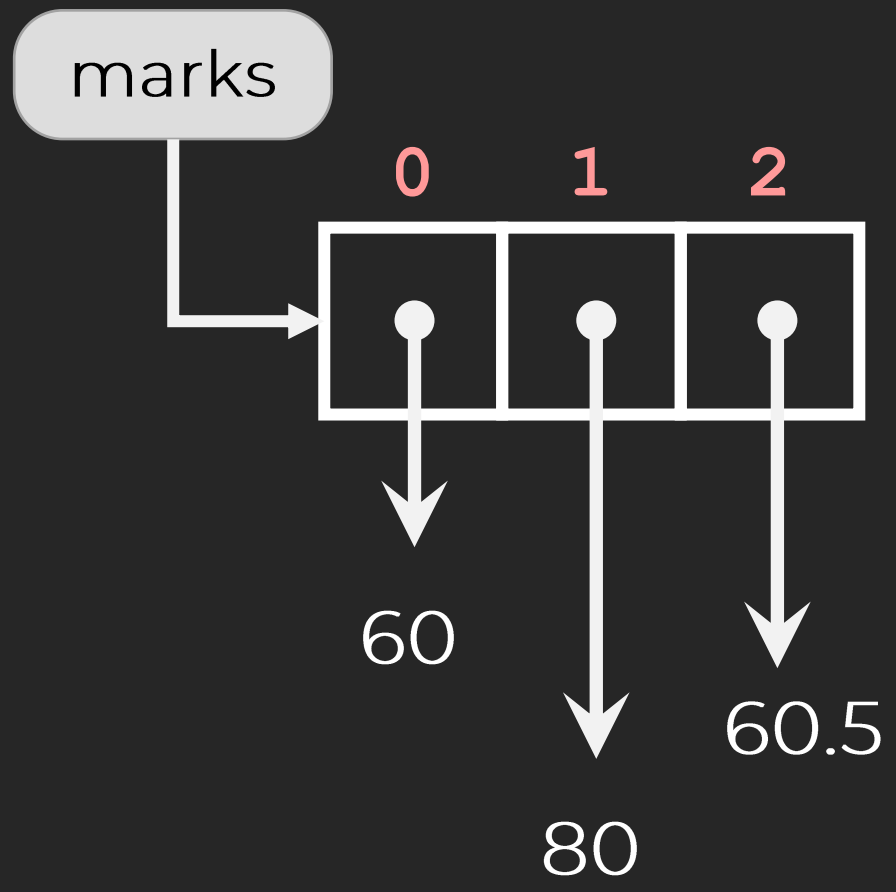
Python shell

```
>>> marks = []
>>> marks.append(60)
>>> marks.append(80)
>>> marks.append(60.5)
>>>
```



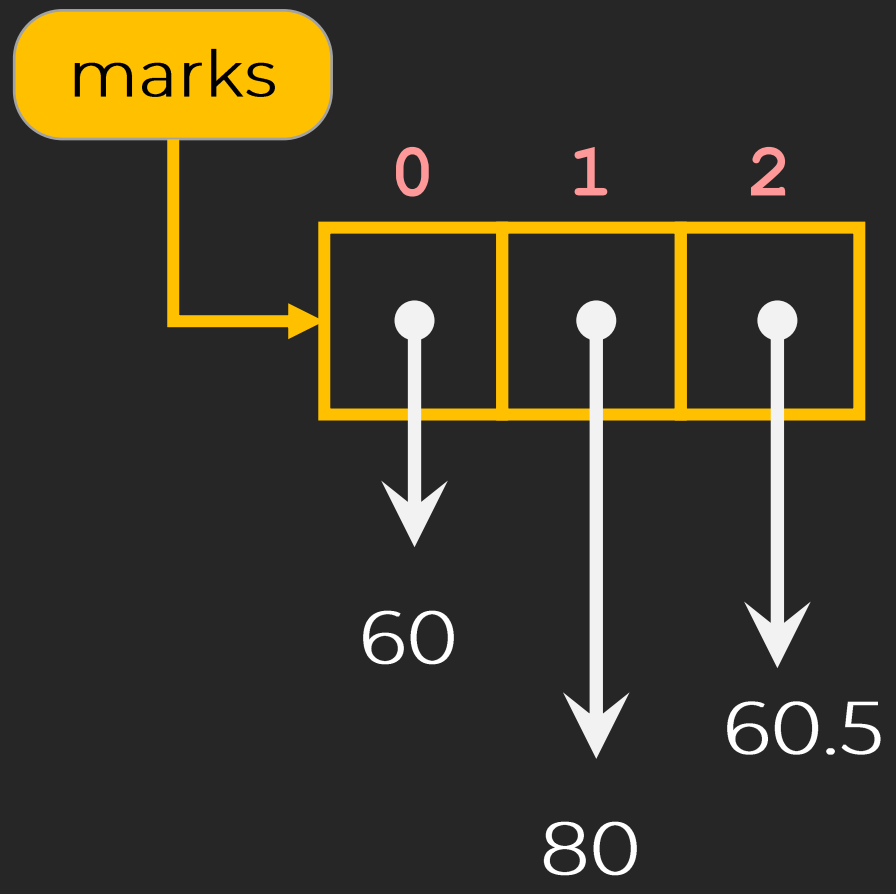
Python shell

```
>>> marks = []
>>> marks.append(60)
>>> marks.append(80)
>>> marks.append(60.5)
>>> print(marks)
```



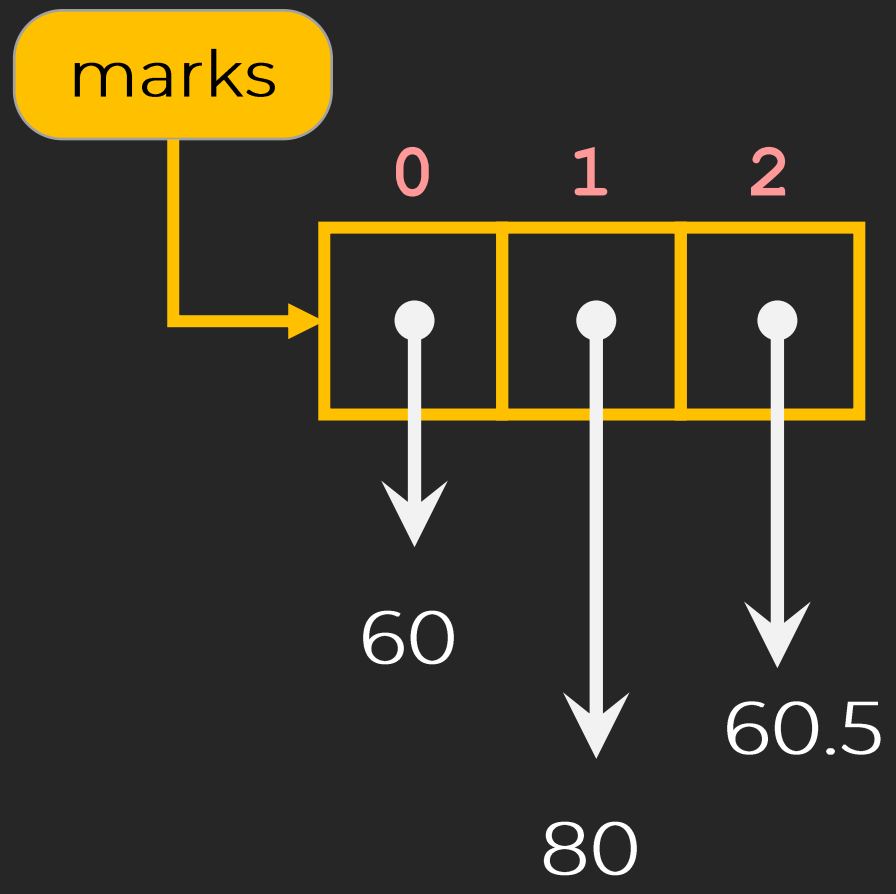
Python shell

```
>>> marks = []
>>> marks.append(60)
>>> marks.append(80)
>>> marks.append(60.5)
>>> print(marks)
```



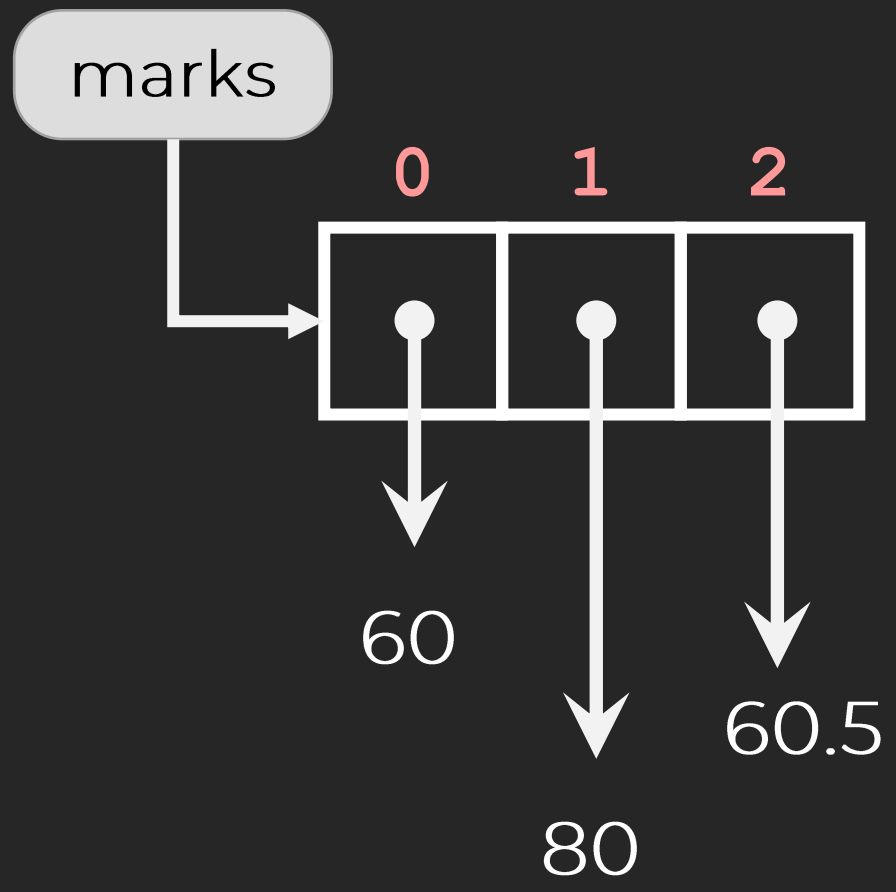
Python shell

```
>>> marks = []
>>> marks.append(60)
>>> marks.append(80)
>>> marks.append(60.5)
>>> print(marks)
[60, 80, 60.5]
>>>
```



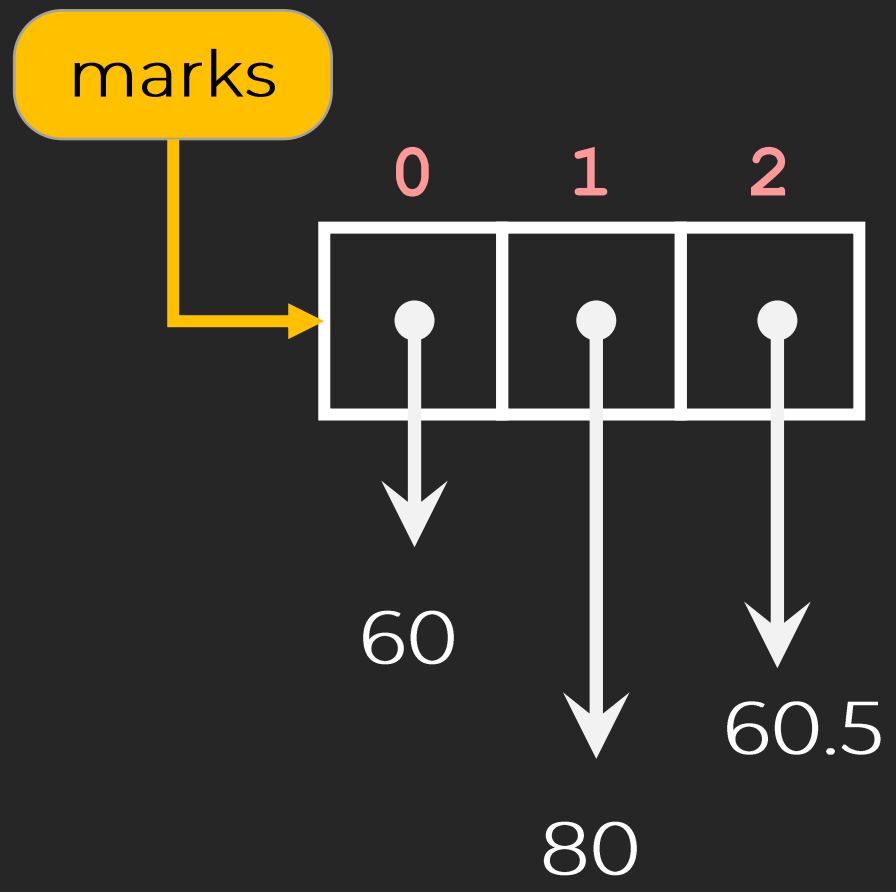
Python shell

```
>>> marks = []
>>> marks.append(60)
>>> marks.append(80)
>>> marks.append(60.5)
>>> print(marks)
[60, 80, 60.5]
>>> print(marks[1])
```



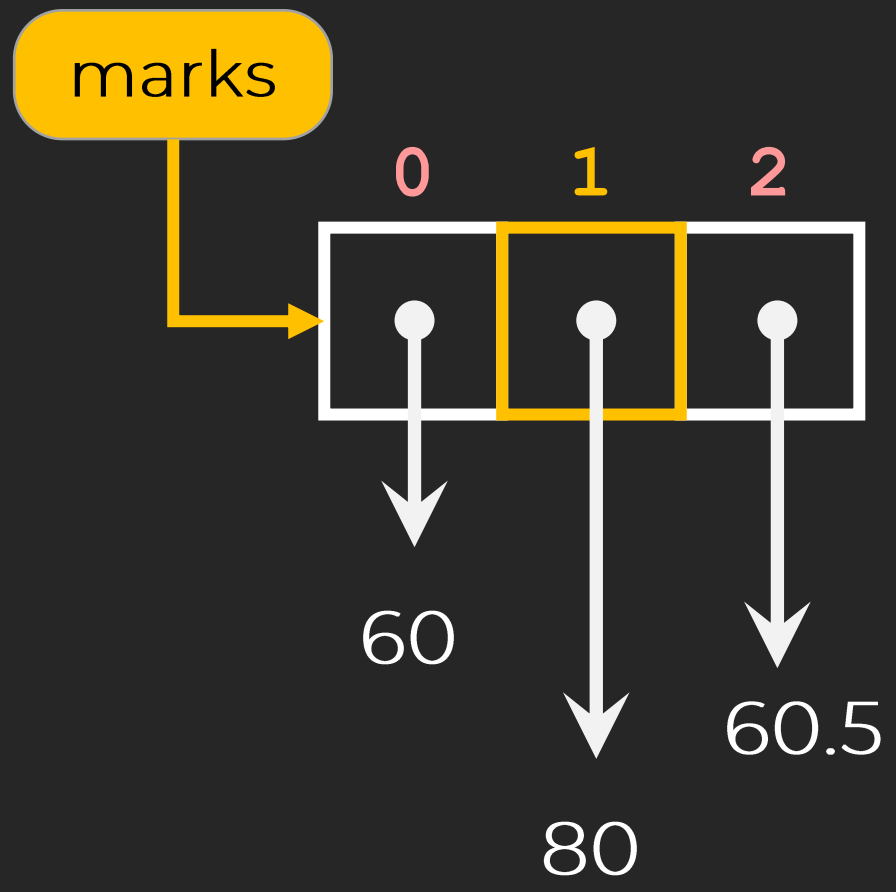
Python shell

```
>>> marks = []
>>> marks.append(60)
>>> marks.append(80)
>>> marks.append(60.5)
>>> print(marks)
[60, 80, 60.5]
>>> print(marks[1])
```



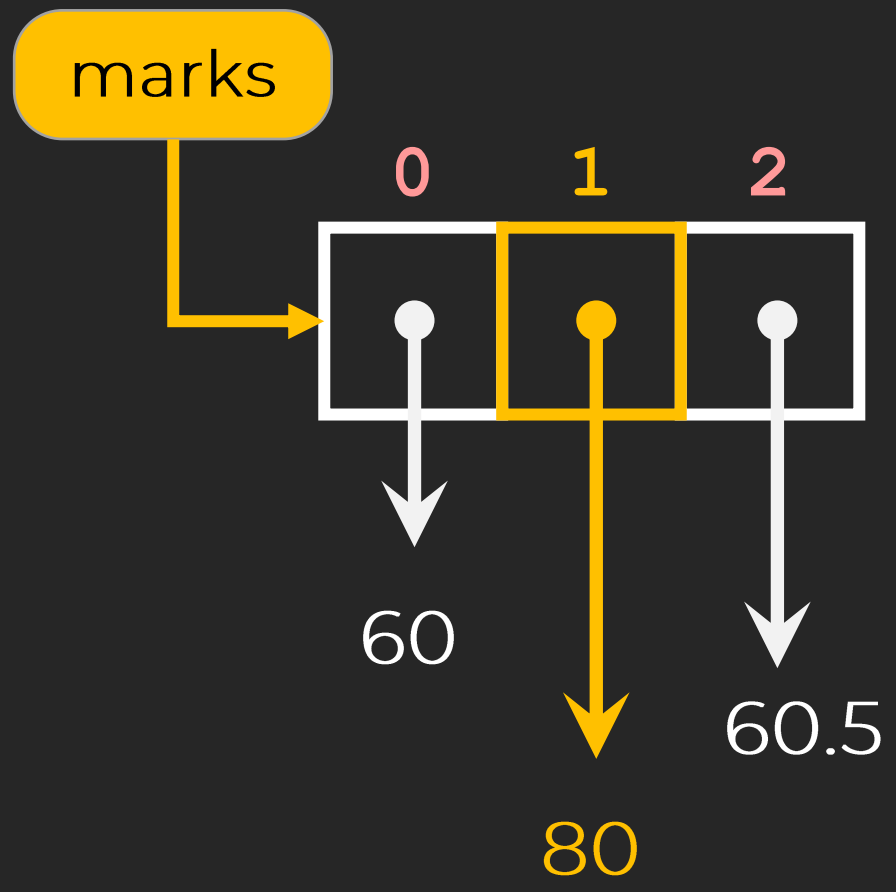
Python shell

```
>>> marks = []
>>> marks.append(60)
>>> marks.append(80)
>>> marks.append(60.5)
>>> print(marks)
[60, 80, 60.5]
>>> print(marks[1])
```



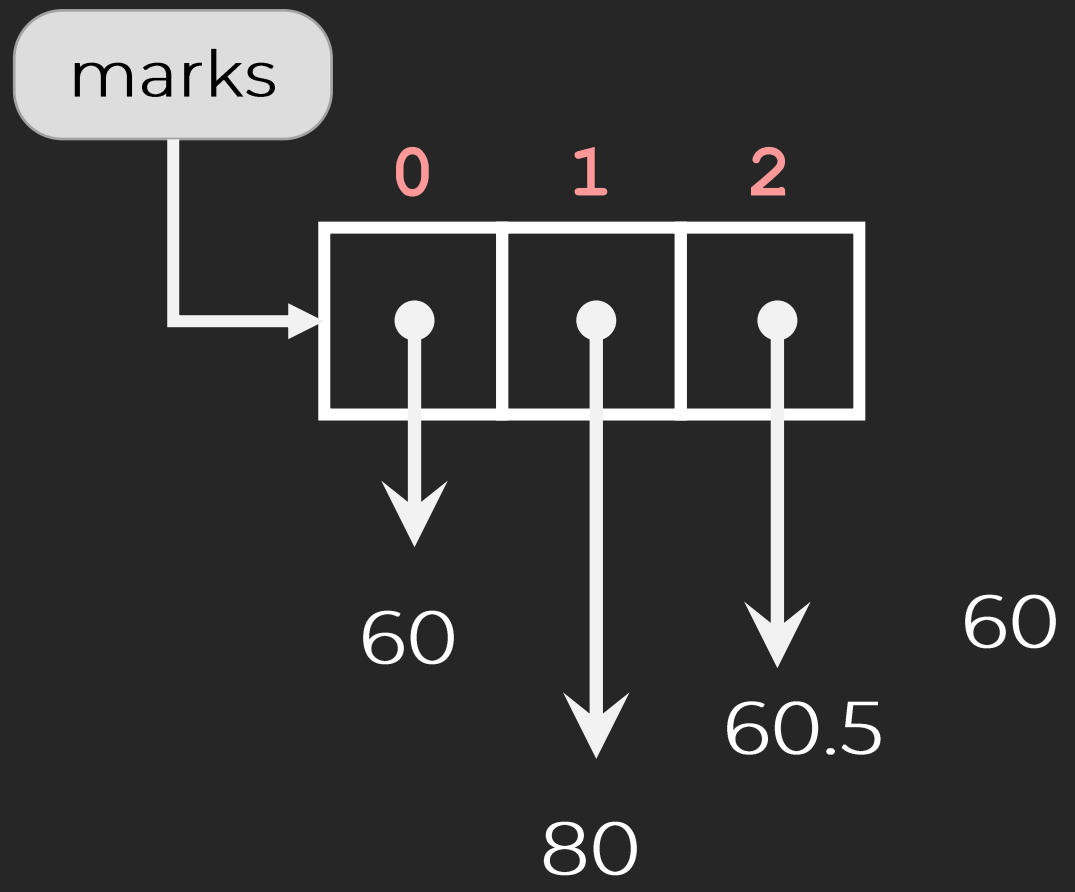
Python shell

```
>>> marks = []
>>> marks.append(60)
>>> marks.append(80)
>>> marks.append(60.5)
>>> print(marks)
[60, 80, 60.5]
>>> print(marks[1])
80
>>>
```



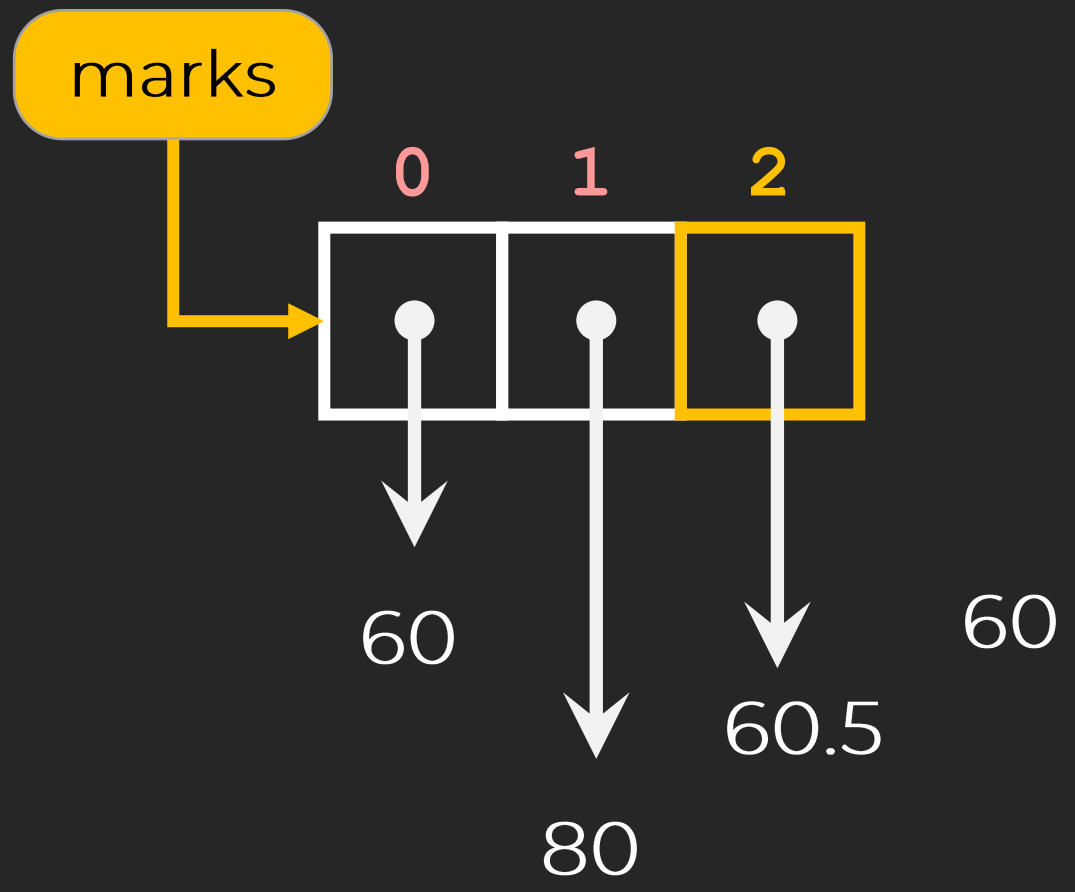
Python shell

```
>>> marks = []
>>> marks.append(60)
>>> marks.append(80)
>>> marks.append(60.5)
>>> print(marks)
[60, 80, 60.5]
>>> print(marks[1])
80
>>> marks[2] = 60
>>>
```



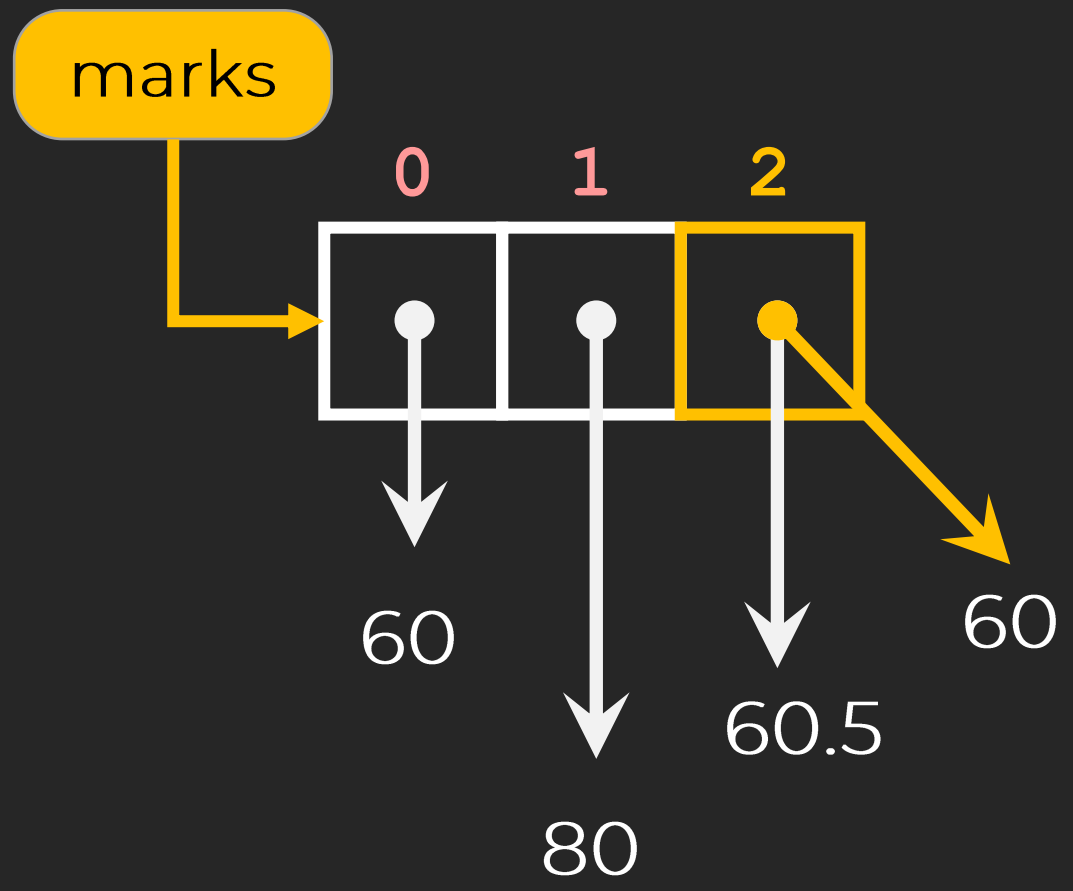
Python shell

```
>>> marks = []
>>> marks.append(60)
>>> marks.append(80)
>>> marks.append(60.5)
>>> print(marks)
[60, 80, 60.5]
>>> print(marks[1])
80
>>> marks[2] = 60
>>>
```



Python shell

```
>>> marks = []
>>> marks.append(60)
>>> marks.append(80)
>>> marks.append(60.5)
>>> print(marks)
[60, 80, 60.5]
>>> print(marks[1])
80
>>> marks[2] = 60
>>>
```



It is very important to remember methods available for a given data structure.

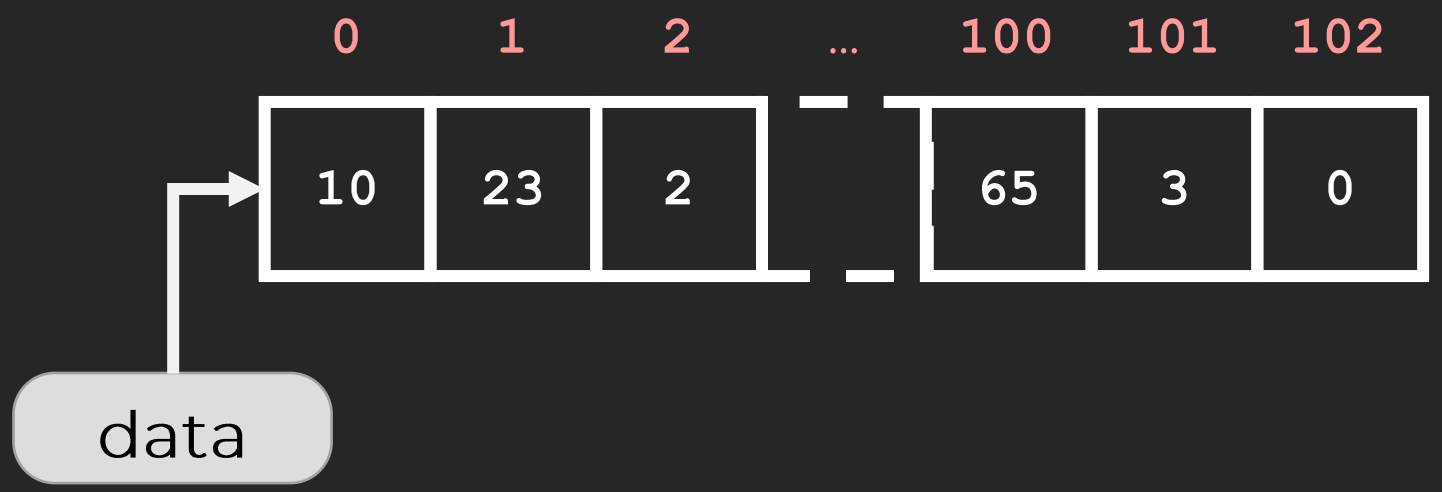
- clear(...) remove all items from L
- count(...) return number of occurrences of value
- extend(...) extend list by appending elements from the iterable
- index(...) return first index of value.
- insert(...) insert object before index
- pop(...) remove and return item at index (default last).
- remove(...) remove first occurrence of value.

In addition to knowing the methods, it is important to know their performances.

Let's consider the method pop() which removes an element from the list.

Python shell

```
>>>
>>>
```

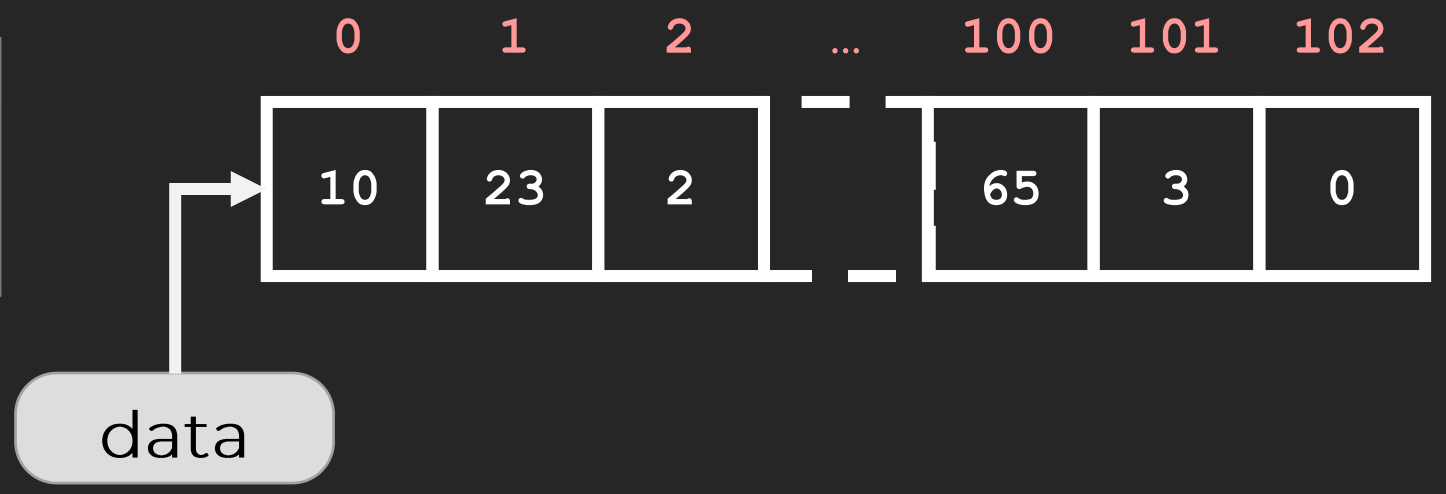


In addition to knowing the methods, it is important to know their performances.

Let's consider the method pop() which removes an element from the list.

Python shell

```
>>> data.pop()
>>>
```

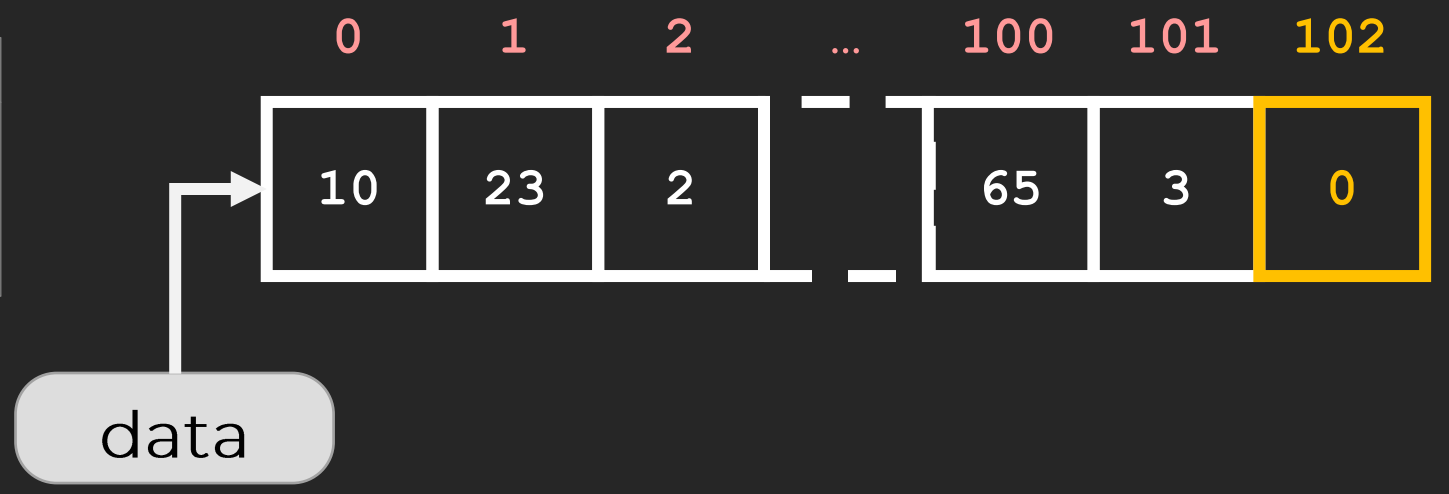


In addition to knowing the methods, it is important to know their performances.

Let's consider the method pop() which removes an element from the list.

Python shell

```
>>> data.pop()
>>>
```

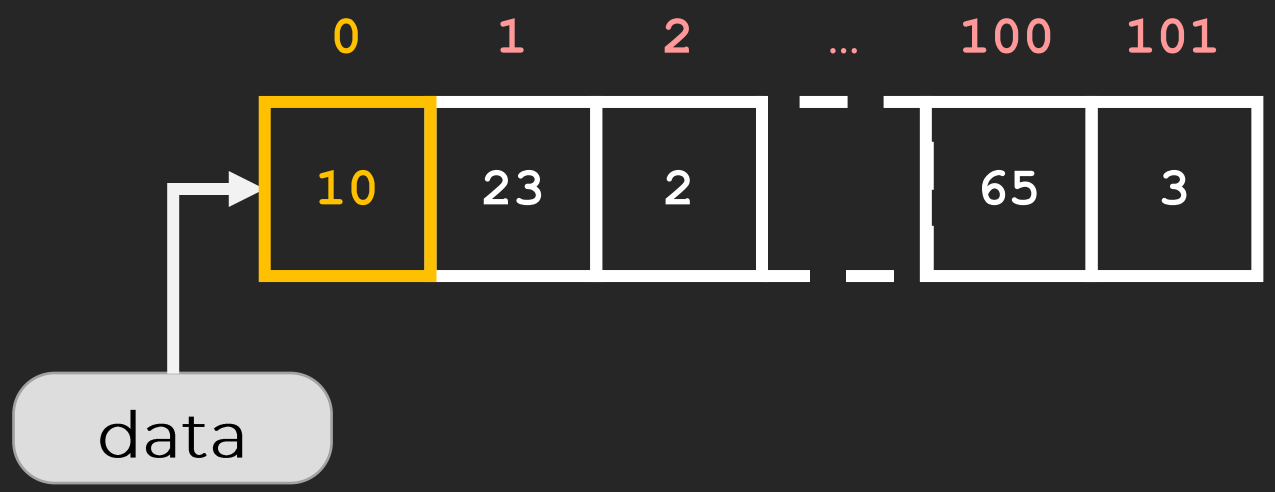


In addition to knowing the methods, it is important to know their performances.

Let's consider the method pop() which removes an element from the list.

Python shell

```
>>> data.pop()
>>> data.pop(0)
>>>
```

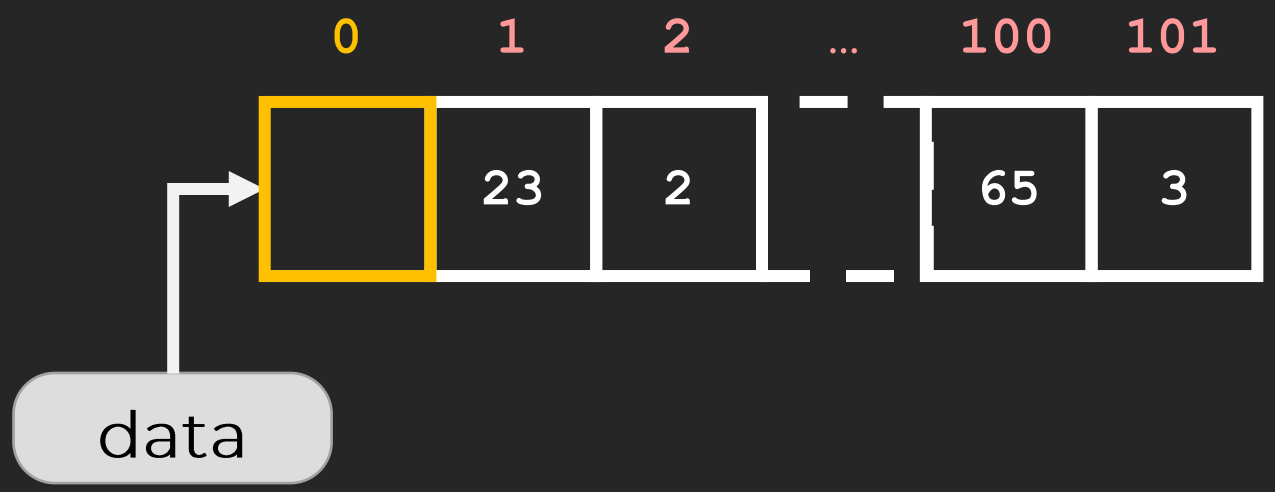


In addition to knowing the methods, it is important to know their performances.

Let's consider the method pop() which removes an element from the list.

Python shell

```
>>> data.pop()
>>> data.pop(0)
>>>
```

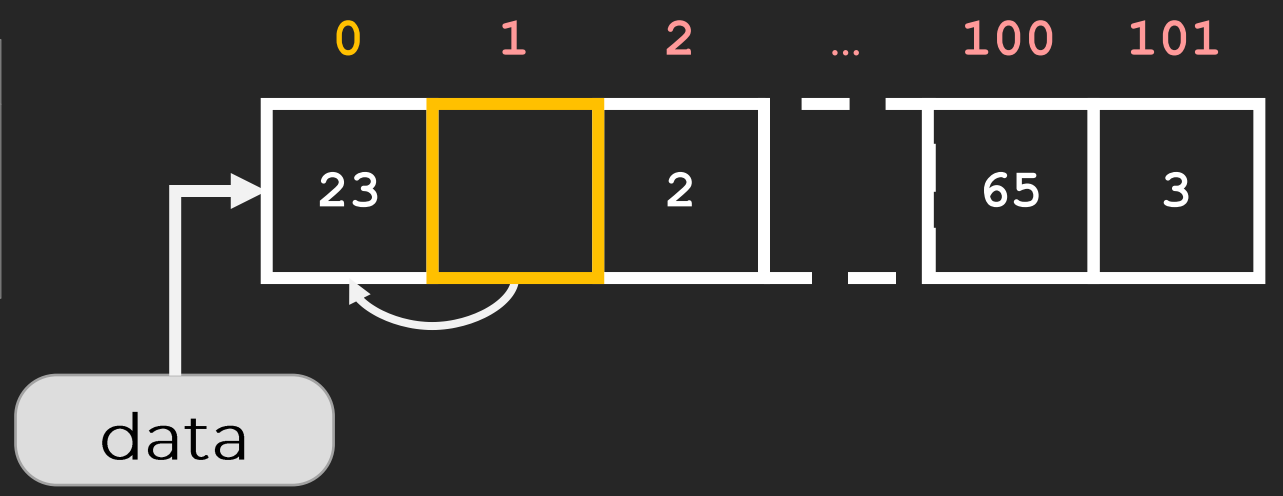


In addition to knowing the methods, it is important to know their performances.

Let's consider the method pop() which removes an element from the list.

Python shell

```
>>> data.pop()
>>> data.pop(0)
>>>
```

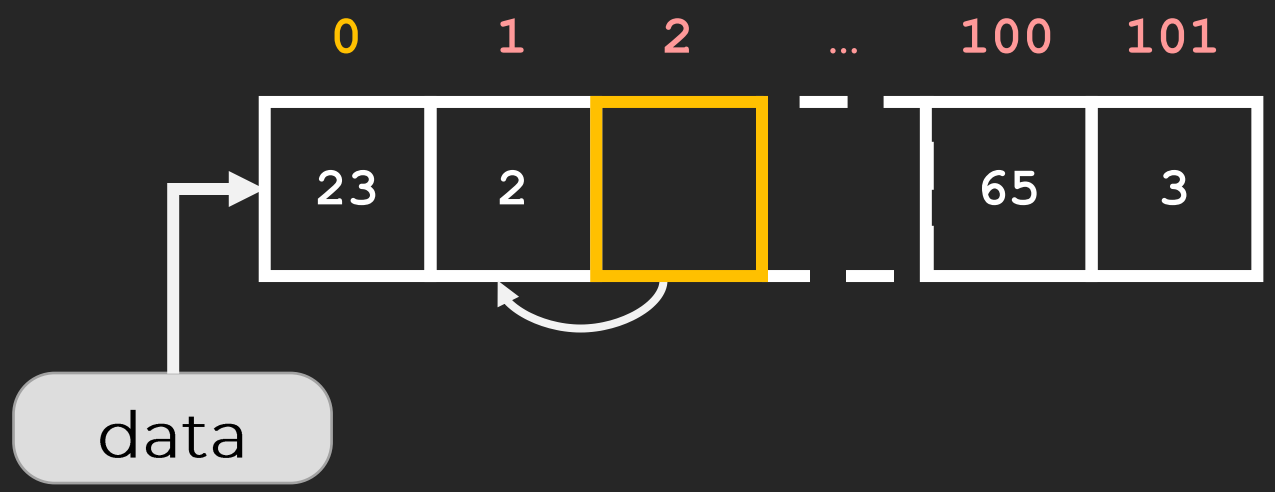


In addition to knowing the methods, it is important to know their performances.

Let's consider the method pop() which removes an element from the list.

Python shell

```
>>> data.pop()
>>> data.pop(0)
>>>
```

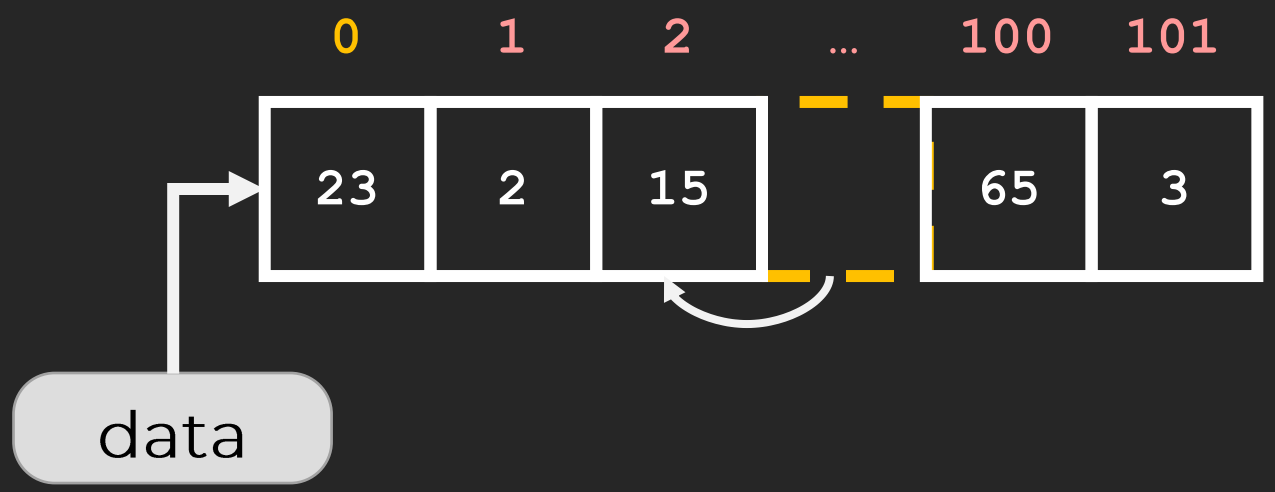


In addition to knowing the methods, it is important to know their performances.

Let's consider the method pop() which removes an element from the list.

Python shell

```
>>> data.pop()
>>> data.pop(0)
>>>
```

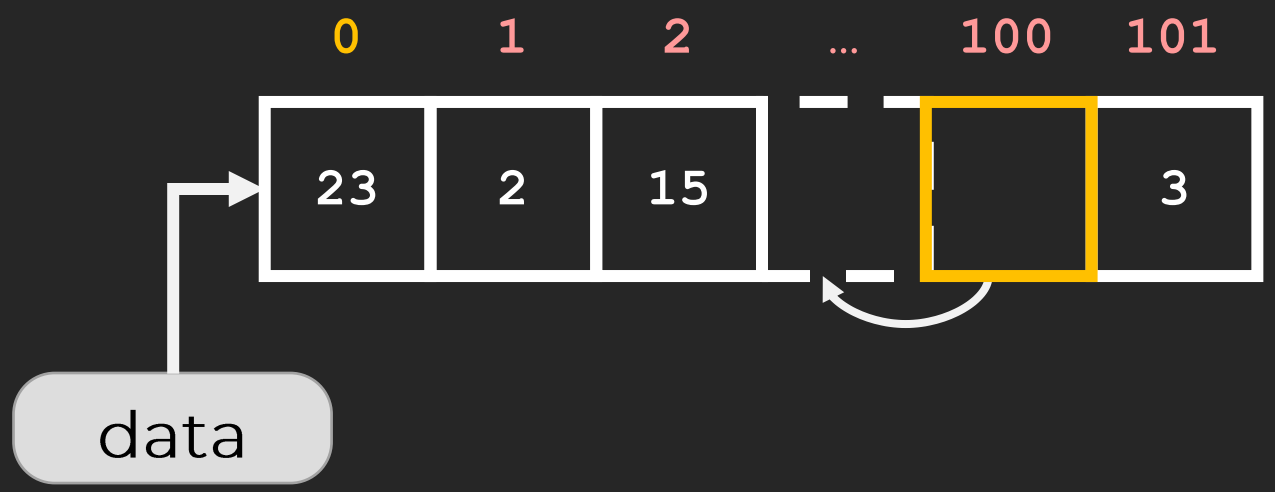


In addition to knowing the methods, it is important to know their performances.

Let's consider the method pop() which removes an element from the list.

Python shell

```
>>> data.pop()
>>> data.pop(0)
>>>
```

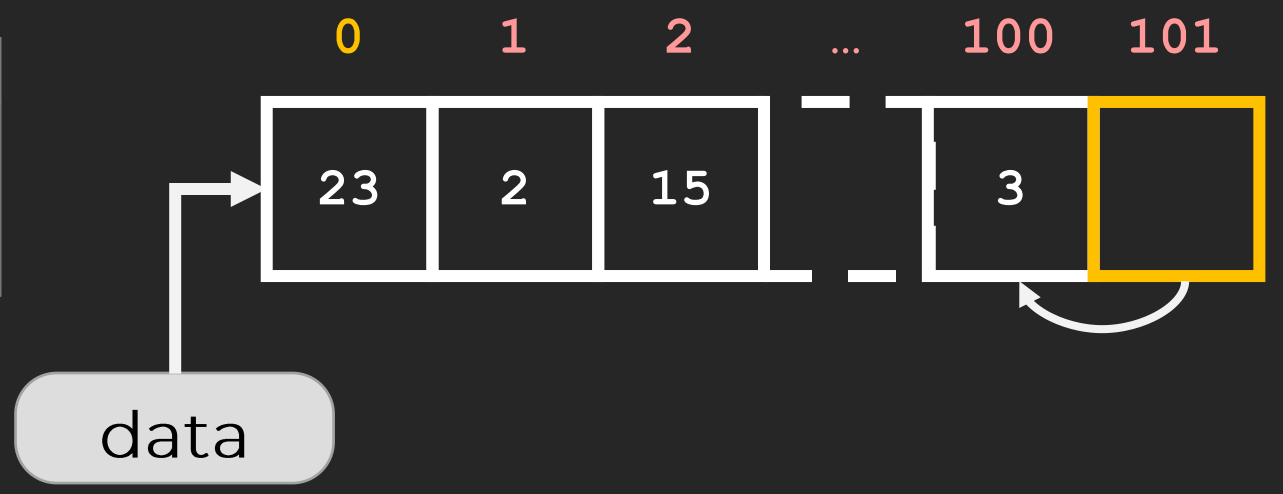


In addition to knowing the methods, it is important to know their performances.

Let's consider the method pop() which removes an element from the list.

Python shell

```
>>> data.pop()
>>> data.pop(0)
>>>
```



Iterating Through Lists

There are two common ways to iterate through an entire list.

A for loop is usually preferred to a while loop.

The variable **grade** will take the value 60, then 80, ...

READ

Python script

```
marks = [60,80,60,75]
for grade in marks:
    print(grade)
```

The variable **index** will take the value 0, then 1, 2, and 3.

READ/ ASSIGN

Python script

```
marks = [60,80,60,75]
for index in range(len(marks)):
    marks[index] *= 1.1
```

If you are unsure that you will need to traverse the entire list, you may want to use a while loop.

Python script

```
marks = [60,80,-60,75]
index = 0
while index < len(marks) and  marks[index] >= 0:
    print(grade)
    index += 1
```

You must initialise index to 0, the position of the first element

You need to increase index to avoid infinite loop

In this video we have seen lists, one of the built-in data structure provided by Python to represent a collection of objects. You will find some similarities with two other data structures, tuples and sets.