# What is a List?

In programming, it is common to want to work with collections of data. In Python, a *list* is one of the many built-in [data structures](https://en.wikipedia.org/wiki/Data_structure) that allows us to work with a collection of data in sequential order.

Suppose we want to make a list of the heights of students in a class:

* Noelle is 61 inches tall
* Ava is 70 inches tall
* Sam is 67 inches tall
* Mia is 64 inches tall

In Python, we can create a variable called heights to store these integers into a [*list*](https://www.codecademy.com/resources/docs/python/lists?page_req=catalog):

heights = [61, 70, 67, 64]

Notice that:

1. A list begins and ends with square brackets ([ and ]).
2. Each item (i.e., 67 or 70) is separated by a comma (,)
3. It’s considered good practice to insert a space () after each comma, but your code will run just fine if you forget the space.

# Data structure

[Graphical user interface, diagram

Description automatically generated](https://en.wikipedia.org/wiki/File:Hash_table_3_1_1_0_1_0_0_SP.svg)

A data structure known as a [hash table](https://en.wikipedia.org/wiki/Hash_table).

In [computer science](https://en.wikipedia.org/wiki/Computer_science), a **data structure** is a [data](https://en.wikipedia.org/wiki/Data) organization, management, and storage format that is usually chosen for [efficient](https://en.wikipedia.org/wiki/Efficiency) [access](https://en.wikipedia.org/wiki/Data_access) to data.[[1]](https://en.wikipedia.org/wiki/Data_structure#cite_note-1)[[2]](https://en.wikipedia.org/wiki/Data_structure#cite_note-2)[[3]](https://en.wikipedia.org/wiki/Data_structure#cite_note-3) More precisely, a data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data,[[4]](https://en.wikipedia.org/wiki/Data_structure#cite_note-4) i.e., it is an [algebraic structure](https://en.wikipedia.org/wiki/Algebraic_structure) about [data](https://en.wikipedia.org/wiki/Data).

## Usage[[edit](https://en.wikipedia.org/w/index.php?title=Data_structure&action=edit&section=1)]

Data structures serve as the basis for [abstract data types](https://en.wikipedia.org/wiki/Abstract_data_type) (ADT). The ADT defines the logical form of the data type. The data structure implements the physical form of the [data type](https://en.wikipedia.org/wiki/Data_type).[[5]](https://en.wikipedia.org/wiki/Data_structure#cite_note-5)

Different types of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks. For example, [relational databases](https://en.wikipedia.org/wiki/Relational_database) commonly use [B-tree](https://en.wikipedia.org/wiki/B-tree) indexes for data retrieval,[[6]](https://en.wikipedia.org/wiki/Data_structure#cite_note-6) while [compiler](https://en.wikipedia.org/wiki/Compiler) [implementations](https://en.wikipedia.org/wiki/Implementation) usually use [hash tables](https://en.wikipedia.org/wiki/Hash_table) to look up identifiers.[[7]](https://en.wikipedia.org/wiki/Data_structure#cite_note-7)

Data structures provide a means to manage large amounts of data efficiently for uses such as large [databases](https://en.wikipedia.org/wiki/Database) and internet indexing services. Usually, efficient data structures are key to designing efficient [algorithms](https://en.wikipedia.org/wiki/Algorithm). Some formal design methods and [programming languages](https://en.wikipedia.org/wiki/Programming_language) emphasize data structures, rather than algorithms, as the key organizing factor in software design. Data structures can be used to organize the storage and retrieval of information stored in both [main memory](https://en.wikipedia.org/wiki/Main_memory) and secondary memory.[[8]](https://en.wikipedia.org/wiki/Data_structure#cite_note-8)

## Implementation[[edit](https://en.wikipedia.org/w/index.php?title=Data_structure&action=edit&section=2)]

Data structures are generally based on the ability of a [computer](https://en.wikipedia.org/wiki/Computer) to fetch and store data at any place in its memory, specified by a [pointer](https://en.wikipedia.org/wiki/Pointer_(computer_programming))—a [bit](https://en.wikipedia.org/wiki/Bit) [string](https://en.wikipedia.org/wiki/String_(computer_science)), representing a [memory address](https://en.wikipedia.org/wiki/Memory_address), that can be itself stored in memory and manipulated by the program. Thus, the [array](https://en.wikipedia.org/wiki/Array_data_structure) and [record](https://en.wikipedia.org/wiki/Record_(computer_science)) data structures are based on computing the addresses of data items with [arithmetic operations](https://en.wikipedia.org/wiki/Arithmetic_operations), while the [linked data structures](https://en.wikipedia.org/wiki/Linked_data_structure) are based on storing addresses of data items within the structure itself.

The implementation of a data structure usually requires writing a set of [procedures](https://en.wikipedia.org/wiki/Subroutine) that create and manipulate instances of that structure. The efficiency of a data structure cannot be analyzed separately from those operations. This observation motivates the theoretical concept of an [abstract data type](https://en.wikipedia.org/wiki/Abstract_data_type), a data structure that is defined indirectly by the operations that may be performed on it, and the mathematical properties of those operations (including their space and time cost).[[9]](https://en.wikipedia.org/wiki/Data_structure#cite_note-9)

## Examples[[edit](https://en.wikipedia.org/w/index.php?title=Data_structure&action=edit&section=3)]

*Main article:*[*List of data structures*](https://en.wikipedia.org/wiki/List_of_data_structures)

[A screenshot of a video game

Description automatically generated](https://en.wikipedia.org/wiki/File:Python_3._The_standard_type_hierarchy.png)

The standard [type](https://en.wikipedia.org/wiki/Data_type) hierarchy of the programming language [Python 3](https://en.wikipedia.org/wiki/Python_(programming_language)).

There are numerous types of data structures, generally built upon simpler [primitive data types](https://en.wikipedia.org/wiki/Primitive_data_type). Well known examples are:[[10]](https://en.wikipedia.org/wiki/Data_structure#cite_note-10)

* An [*array*](https://en.wikipedia.org/wiki/Array_data_structure) is a number of elements in a specific order, typically all of the same type (depending on the language, individual elements may either all be forced to be the same type, or may be of almost any type). Elements are accessed using an integer index to specify which element is required. Typical implementations allocate contiguous memory words for the elements of arrays (but this is not always a necessity). Arrays may be fixed-length or resizable.
* A [*linked list*](https://en.wikipedia.org/wiki/Linked_list) (also just called *list*) is a linear collection of data elements of any type, called nodes, where each node has itself a value, and points to the next node in the linked list. The principal advantage of a linked list over an array is that values can always be efficiently inserted and removed without relocating the rest of the list. Certain other operations, such as [random access](https://en.wikipedia.org/wiki/Random_access) to a certain element, are however slower on lists than on arrays.
* A [*record*](https://en.wikipedia.org/wiki/Record_(computer_science)) (also called *tuple* or *struct*) is an [aggregate data](https://en.wikipedia.org/wiki/Aggregate_data) structure. A record is a value that contains other values, typically in fixed number and sequence and typically indexed by names. The elements of records are usually called *fields* or *members*. In the context of [object-oriented programming](https://en.wikipedia.org/wiki/Object-oriented_programming), records are known as [plain old data structures](https://en.wikipedia.org/wiki/Plain_old_data_structure) to distinguish them from objects.[[11]](https://en.wikipedia.org/wiki/Data_structure#cite_note-11)
* [*Hash tables*](https://en.wikipedia.org/wiki/Hash_table), [*graphs*](https://en.wikipedia.org/wiki/Graph_(computer_science)) and [*binary trees*](https://en.wikipedia.org/wiki/Binary_trees).

## Language support[[edit](https://en.wikipedia.org/w/index.php?title=Data_structure&action=edit&section=4)]

Most [assembly languages](https://en.wikipedia.org/wiki/Assembly_language) and some [low-level languages](https://en.wikipedia.org/wiki/Low-level_programming_language), such as [BCPL](https://en.wikipedia.org/wiki/BCPL) (Basic Combined Programming Language), lack built-in support for data structures. On the other hand, many [high-level programming languages](https://en.wikipedia.org/wiki/High-level_programming_language) and some higher-level assembly languages, such as [MASM](https://en.wikipedia.org/wiki/MASM), have special syntax or other built-in support for certain data structures, such as records and arrays. For example, the [C](https://en.wikipedia.org/wiki/C_(programming_language)) (a direct descendant of BCPL) and [Pascal](https://en.wikipedia.org/wiki/Pascal_(programming_language)) languages support [structs](https://en.wikipedia.org/wiki/Record_(computer_science)) and records, respectively, in addition to vectors (one-dimensional [arrays](https://en.wikipedia.org/wiki/Array_data_type)) and multi-dimensional arrays.[[12]](https://en.wikipedia.org/wiki/Data_structure#cite_note-gnu-c-12)[[13]](https://en.wikipedia.org/wiki/Data_structure#cite_note-13)

Most programming languages feature some sort of [library](https://en.wikipedia.org/wiki/Library_(computing)) mechanism that allows data structure implementations to be reused by different programs. Modern languages usually come with standard libraries that implement the most common data structures. Examples are the [C++](https://en.wikipedia.org/wiki/C%2B%2B) [Standard Template Library](https://en.wikipedia.org/wiki/Standard_Template_Library), the [Java Collections Framework](https://en.wikipedia.org/wiki/Java_Collections_Framework), and the [Microsoft](https://en.wikipedia.org/wiki/Microsoft) [.NET Framework](https://en.wikipedia.org/wiki/.NET_Framework).

Modern languages also generally support [modular programming](https://en.wikipedia.org/wiki/Modular_programming), the separation between the [interface](https://en.wikipedia.org/wiki/Interface_(computing)) of a library module and its implementation. Some provide [opaque data types](https://en.wikipedia.org/wiki/Opaque_data_type) that allow clients to hide implementation details. [Object-oriented programming languages](https://en.wikipedia.org/wiki/Object-oriented_programming_language), such as [C++](https://en.wikipedia.org/wiki/C%2B%2B), [Java](https://en.wikipedia.org/wiki/Java_(programming_language)), and [Smalltalk](https://en.wikipedia.org/wiki/Smalltalk), typically use [classes](https://en.wikipedia.org/wiki/Classes_(computer_science)) for this purpose.

Many known data structures have [concurrent](https://en.wikipedia.org/wiki/Concurrent_data_structure) versions which allow multiple computing threads to access a single concrete instance of a data structure simultaneously.[[14]](https://en.wikipedia.org/wiki/Data_structure#cite_note-14)

# What can a List contain?

Lists can contain more than just numbers.

Let’s revisit our classroom example with heights:

* Noelle is 61 inches tall
* Ava is 70 inches tall
* Sam is 67 inches tall
* Mia is 64 inches tall

Instead of storing each student’s height, we can make a list that contains their names:

names = ["Noelle", "Ava", "Sam", "Mia"]

We can even combine multiple [data types](https://www.codecademy.com/resources/docs/python/data-types?page_ref=catalog) in one list. For example, this list contains both a [string](https://www.codecademy.com/resources/docs/python/strings?page_req=catalog) and an integer:

mixed\_list\_string\_number = ["Noelle", 61]

Lists can contain any data type in Python! For example, this list contains a string, integer, boolean, and float.

mixed\_list\_common = ["Mia", 27, False, 0.5]

Let’s experiment with different data types in our own lists!

**Empty Lists**

A list doesn’t have to contain anything. You can create an empty list like this:

empty\_list = []

Why would we create an empty list?

Usually, it’s because we’re planning on filling it up later based on some other input. We’ll talk about two ways of filling up a list in the next exercise.

**List Methods**

As we start exploring lists further in the next exercises, we will encounter the concept of a *method*.

In Python, for any specific data-type ( strings, booleans, lists, etc. ) there is built-in functionality that we can use to create, manipulate, and even delete our data. We call this built-in functionality a method.

For lists, methods will follow the form of list\_name.method(). Some methods will require an input value that will go between the parenthesis of the method ( ).

An example of a popular list method is [.append()](https://www.codecademy.com/resources/docs/python/lists/append?page_req=catalog), which allows us to add an element to the end of a list.

append\_example = [ 'This', 'is', 'an', 'example']  
append\_example.append('list')  
   
print(append\_example)

Will output:

['This', 'is', 'an', 'example', 'list']

**Accessing List Elements**

We are interviewing candidates for a job. We will call each candidate in order, represented by a Python list:

calls = ["Juan", "Zofia", "Amare", "Ezio", "Ananya"]

First, we’ll call "Juan", then "Zofia", etc.

In Python, we call the location of an element in a list its *index*.

Python lists are *zero-indexed*. This means that the first element in a list has index 0, rather than 1.

Here are the index numbers for the list calls:

| **Element** | **Index** |
| --- | --- |
| "Juan" | 0 |
| "Zofia" | 1 |
| "Amare" | 2 |
| "Ezio" | 3 |
| "Ananya" | 4 |

In this example, the element with *index* 2 is "Amare".

We can select a single element from a list by using square brackets ([]) and the index of the list item. If we wanted to select the third element from the list, we’d use calls[2]:

print(calls[2])

Will output:

Amare

**Note:** When accessing elements of a list, you *must* use an int as the index. If you use a float, you will get an error. This can be especially tricky when using division. For example print(calls[4/2]) will result in an error, because 4/2 gets evaluated to the float 2.0.

To solve this problem, you can force the result of your division to be an int by using the int() function. int() takes a number and cuts off the decimal point. For example, int(5.9) and int(5.0) will both become 5. Therefore, calls[int(4/2)] will result in the same value as calls[2], whereas calls[4/2] will result in an error.

**Accessing List Elements: Negative Index**

What if we want to select the last element of a list?

We can use the index -1 to select the last item of a list, even when we don’t know how many elements are in a list.

Consider the following list with 6 elements:

pancake\_recipe = ["eggs", "flour", "butter", "milk", "sugar", "love"]

If we select the -1 index, we get the final element, "love".

print(pancake\_recipe[-1])

Would output:

love

This is equivalent to selecting the element with index 5:

print(pancake\_recipe[5])

Would output:

love

Here are the negative index numbers for our list:

| **Element** | **Index** |
| --- | --- |
| "eggs" | -6 |
| "flour" | -5 |
| "butter" | -4 |
| "milk" | -3 |
| "sugar" | -2 |
| "love" | -1 |

**Modifying List Elements**

Let’s return to our garden.

garden = ["Tomatoes", "Green Beans", "Cauliflower", "Grapes"]

Unfortunately, we forgot to water our cauliflower and we don’t think it is going to recover.

Thankfully our friend Jiho from Petal Power came to the rescue. Jiho gifted us some strawberry seeds. We will replace the cauliflower with our new seeds.

We will need to modify the list to accommodate the change to our garden list. To change a value in a list, reassign the value using the specific index.

garden[2] = "Strawberries"  
   
print(garden)

Will output:

["Tomatoes", "Green Beans", "Strawberries", "Grapes"]

Negative indices will work as well.

garden[-1] = "Raspberries"  
   
print(garden)

Will output:

["Tomatoes", "Green Beans", "Strawberries", "Raspberries"]

**Shrinking a List: Remove**

We can remove elements in a list using the [.remove()](https://www.codecademy.com/resources/docs/python/lists/remove?page_req=catalog) Python method.

Suppose we have a filled list called shopping\_line that represents a line at a grocery store:

shopping\_line = ["Cole", "Kip", "Chris", "Sylvana"]

We could remove "Chris" by using the .remove() method:

shopping\_line.remove("Chris")  
   
print(shopping\_line)

If we examine shopping\_line, we can see that it now doesn’t contain "Chris":

["Cole", "Kip", "Sylvana"]

We can also use .remove() on a list that has duplicate elements.

Only the first instance of the matching element is removed:

# Create a list  
shopping\_line = ["Cole", "Kip", "Chris", "Sylvana", "Chris"]  
   
# Remove a element  
shopping\_line.remove("Chris")  
print(shopping\_line)

Will output:

["Cole", "Kip", "Sylvana", "Chris"]

**Two-Dimensional (2D) Lists**

We’ve seen that the items in a list can be numbers or strings. Lists can contain other lists! We will commonly refer to these as *two-dimensional (2D)* lists.

Once more, let’s look at a class height example:

* Noelle is 61 inches tall
* Ava is 70 inches tall
* Sam is 67 inches tall
* Mia is 64 inches tall

Previously, we saw that we could create a list representing both Noelle’s name and height:

noelle = ["Noelle", 61]

We can put several of these lists into one list, such that each entry in the list represents a student and their height:

heights = [["Noelle", 61], ["Ava", 70], ["Sam", 67], ["Mia", 64]]

We will often find that a two-dimensional list is a very good structure for representing grids such as games like tic-tac-toe.

#A 2d list with three lists in each of the indices.   
tic\_tac\_toe = [  
            ["X","O","X"],   
            ["O","X","O"],   
            ["O","O","X"]  
]

**Accessing 2D Lists**

Let’s return to our classroom heights example:

heights = [["Noelle", 61], ["Ali", 70], ["Sam", 67]]

Two-dimensional lists can be accessed similar to their one-dimensional counterpart. Instead of providing a single pair of brackets [ ] we will use an additional set for each dimension past the first.

If we wanted to access "Noelle"‘s height:

#Access the sublist at index 0, and then access the 1st index of that sublist.   
noelles\_height = heights[0][1]   
print(noelles\_height)

Would output:

61

Here are the index numbers to access data for the list heights:

| **Element** | **Index** |
| --- | --- |
| "Noelle" | heights[0][0] |
| 61 | heights[0][1] |
| "Ali" | heights[1][0] |
| 70 | heights[1][1] |
| "Sam" | heights[2][0] |
| 67 | heights[2][1] |

Let’s practice accessing data in a two-dimensional list.

**Modifying 2D Lists**

Now that we know how to access two-dimensional lists, modifying the elements should come naturally.

Let’s return to a classroom example, but now instead of heights or test scores, our list stores the student’s favorite hobby!

class\_name\_hobbies = [["Jenny", "Breakdancing"], ["Alexus", "Photography"], ["Grace", "Soccer"]]

"Jenny" changed their mind and is now more interested in "Meditation".

We will need to modify the list to accommodate the change to our class\_name\_hobbies list. To change a value in a two-dimensional list, reassign the value using the specific index.

# The list of Jenny is at index 0. The hobby is at index 1.   
class\_name\_hobbies[0][1] = "Meditation"  
print(class\_name\_hobbies)

Would output:

[["Jenny", "Meditation"], ["Alexus", "Photography"], ["Grace", "Soccer"]]

Negative indices will work as well.

# The list of Grace is the last entry. The hobby is the last element.   
class\_name\_hobbies[-1][-1] = "Football"  
print(class\_name\_hobbies)

Would output:

[["Jenny", "Meditation"], ["Alexus", "Photography"], ["Grace", "Football"]]

**Review**

So far, we have learned:

* How to create a list
* How to access, add, remove, and modify list elements
* How to create a two-dimensional list
* How to access and modify two-dimensional list elements

Let’s practice these skills.