# 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

**Working with Lists**

Now that we know how to create and access list data, we can start to explore additional ways of working with lists.

In this lesson, you’ll learn how to:

* Add and remove items from a list using a specific index.
* Create lists with continuous values.
* Get the length of a list.
* Select portions of a list (called slicing).
* Count the number of times that an element appears in a list.
* Sort a list of items.

**Note:** In some of the exercises, we will be using [built-in functions](https://www.codecademy.com/resources/docs/python/built-in-functions?page_req=catalog) in Python. If you haven’t yet explored the concept of a function, it may look a bit new. Below we compare it to the method syntax we learned in the earlier lesson.

Here is a preview:

# Example syntax for methods  
list.method(input)  
   
# Example syntax for a built-in function   
builtinfuncion(input)

Text

Description automatically generated

**Built-in Functions**

The Python interpreter has a set of functions and types built into it (pre-defined). They are always ready at your disposal; you can use them without needing to import a library.

There are 68 built-in functions and they are listed here in alphabetical order.

**Built-in Functions**

[abs()](https://www.codecademy.com/resources/docs/python/built-in-functions/abs)

Returns the absolute value of a numeric argument.

[all()](https://www.codecademy.com/resources/docs/python/built-in-functions/all)

Returns True if every item in an iterable evaluates to True, otherwise, it returns False.

[any()](https://www.codecademy.com/resources/docs/python/built-in-functions/any)

Takes in an iterable object such as a list or tuple and returns True if any of the elements in the iterable are True. If none of the elements in the iterable are True, returns False.

[ascii()](https://www.codecademy.com/resources/docs/python/built-in-functions/ascii)

Receives as input an object containing string data, and returns the object as a printable representation with escapes for non-ASCII characters (accented characters).

[bin()](https://www.codecademy.com/resources/docs/python/built-in-functions/bin)

Converts an integer into its binary equivalent string.

[bool()](https://www.codecademy.com/resources/docs/python/built-in-functions/bool)

Converts a value to a Boolean True or False value.

[breakpoint()](https://www.codecademy.com/resources/docs/python/built-in-functions/breakpoint)

Engages, configures, and changes the debugger program used in a script.

[bytearray()](https://www.codecademy.com/resources/docs/python/built-in-functions/bytearray)

Returns an array of the given bytes of an object.

[bytes()](https://www.codecademy.com/resources/docs/python/built-in-functions/bytes)

Returns a byte immutable object representing the given bytes of an object.

[callable()](https://www.codecademy.com/resources/docs/python/built-in-functions/callable)

Returns True if an object is callable, and False if an object is not callable.

[chr()](https://www.codecademy.com/resources/docs/python/built-in-functions/chr)

Returns Unicode characters represented by integers ranging between 0 and 1,114,111.

[classmethod()](https://www.codecademy.com/resources/docs/python/built-in-functions/classmethod)

Converts a given function into a class method.

[compile()](https://www.codecademy.com/resources/docs/python/built-in-functions/compile)

Returns a runnable code object created from a string.

[complex()](https://www.codecademy.com/resources/docs/python/built-in-functions/complex)

Converts a given string into a complex number.

[delattr()](https://www.codecademy.com/resources/docs/python/built-in-functions/delattr)

Allows the user to delete attributes from an object.

[dict()](https://www.codecademy.com/resources/docs/python/built-in-functions/dict)

Initializes a new dictionary from mapping n-number of object (key, value) pairs.

[eval()](https://www.codecademy.com/resources/docs/python/built-in-functions/eval)

Returns the value of a Python expression passed as a string.

[filter()](https://www.codecademy.com/resources/docs/python/built-in-functions/filter)

Returns a filter object that applies a function to each item in an iterable and returns the values that are True.

[float()](https://www.codecademy.com/resources/docs/python/built-in-functions/float)

Returns a float value based on a string, numeric data type, or no value at all.

[frozenset()](https://www.codecademy.com/resources/docs/python/built-in-functions/frozenset)

Returns a new frozenset using an optional iterable object such as a string or list.

[hasattr()](https://www.codecademy.com/resources/docs/python/built-in-functions/hasattr)

Returns True if an object has an attribute and False otherwise.

[help()](https://www.codecademy.com/resources/docs/python/built-in-functions/help)

Displays documentation of an object using the Python help utility.

[input()](https://www.codecademy.com/resources/docs/python/built-in-functions/input)

Prompts the user for data and returns it as a string.

[int()](https://www.codecademy.com/resources/docs/python/built-in-functions/int)

Takes in a value that can be converted into an integer, and returns a copy of the value in the int datatype.

[len()](https://www.codecademy.com/resources/docs/python/built-in-functions/len)

Returns the length of an object, which can either be a sequence or collection.

[list()](https://www.codecademy.com/resources/docs/python/built-in-functions/list)

Returns a list from an iterable.

[map()](https://www.codecademy.com/resources/docs/python/built-in-functions/map)

Returns an iterator that takes a function and applies it to every item in an iterable.

[max()](https://www.codecademy.com/resources/docs/python/built-in-functions/max)

Returns the highest value from values given or an iterable.

[min()](https://www.codecademy.com/resources/docs/python/built-in-functions/min)

Returns the lowest value from values given or an iterable.

[next()](https://www.codecademy.com/resources/docs/python/built-in-functions/next)

Returns the next element from an iterable object.

[open()](https://www.codecademy.com/resources/docs/python/built-in-functions/open)

Used for opening files in a Python program.

[pow()](https://www.codecademy.com/resources/docs/python/built-in-functions/pow)

Returns the value of a base number x to the power of an exponent y, with an optional modulus z.

[print()](https://www.codecademy.com/resources/docs/python/built-in-functions/print)

Prints the string representation of an object.

[range()](https://www.codecademy.com/resources/docs/python/built-in-functions/range)

Returns a sequence of numbers based on the given range

[reversed()](https://www.codecademy.com/resources/docs/python/built-in-functions/reversed)

Takes in an iterator object, such as a list or string, and returns a reversed iterator object.

[round()](https://www.codecademy.com/resources/docs/python/built-in-functions/round)

Takes a number and an integer as parameters, and returns the number with decimal places equal to the integer.

[set()](https://www.codecademy.com/resources/docs/python/built-in-functions/set)

Returns a new set based on an optional iterable object such as a list.

[sorted()](https://www.codecademy.com/resources/docs/python/built-in-functions/sorted)

Takes in an iterator object, such as a list, tuple, dictionary, set, or string, and sorts it according to a parameter.

[str()](https://www.codecademy.com/resources/docs/python/built-in-functions/str)

Takes in a value that can be converted into a string, and returns a copy of the value in the string datatype.

[super()](https://www.codecademy.com/resources/docs/python/built-in-functions/super)

Returns a temporary object that allows a given class to inherit the methods and properties of a parent or sibling class.

[tuple()](https://www.codecademy.com/resources/docs/python/built-in-functions/tuple)

Creates a new tuple.

[type()](https://www.codecademy.com/resources/docs/python/built-in-functions/type)

Returns the data type of the argument passed to the function.

[zip()](https://www.codecademy.com/resources/docs/python/built-in-functions/zip)

Takes multiple iterators as input and returns a single zip object made up of a list of tuples.

**Adding by Index: Insert**

The Python list method .insert() allows us to add an element to a specific index in a list.

The [.insert()](https://www.codecademy.com/resources/docs/python/lists/insert?page_ref=catalog) method takes in two inputs:

1. The index you want to insert into.
2. The element you want to insert at the specified index.

The .insert() method will handle shifting over elements and can be used with negative indices.

To see it in action let’s imagine we have a list representing a line at a store:

store\_line = ["Karla", "Maxium", "Martim", "Isabella"]

"Maxium" saved a spot for his friend "Vikor" and we need to adjust the list to add him into the line right behind "Maxium".

For this example, we can assume that "Karla" is the front of the line and the rest of the elements are behind her.

Here is how we would use the .insert() method to insert "Vikor" :

store\_line.insert(2, "Vikor")  
print(store\_line)

Would output:

['Karla', 'Maxium', 'Vikor', 'Martim', 'Isabella']

Some important things to note:

1. The order and number of the inputs is important. The .insert() method expects two inputs, the first being a numerical index, followed by any value as the second input.
2. When we insert an element into a list, all elements from the specified index and up to the last index are shifted one index to the right. This does not apply to inserting an element to the very end of a list as it will simply add an additional index and no other elements will need to shift.

**Removing by Index: Pop**

Just as we learned to insert elements at specific indices, Python gives us a method to remove elements at a specific index using a method called .pop().

The [.pop()](https://www.codecademy.com/resources/docs/python/lists/pop?page_req=catalog) method takes an optional single input:

1. The index for the element you want to remove.

To see it in action, let’s consider a list called cs\_topics that stores a collection of topics one might study in a computer science program.

cs\_topics = ["Python", "Data Structures", "Balloon Making", "Algorithms", "Clowns 101"]

Two of these topics don’t look like they belong, let’s see how we remove them using .pop().

First let’s remove "Clowns 101":

removed\_element = cs\_topics.pop()  
print(cs\_topics)  
print(removed\_element)

Would output:

['Python', 'Data Structures', 'Balloon Making', 'Algorithms']  
'Clowns 101'

Notice two things about this example:

1. The method can be called without a specific index. Using .pop() without an index will remove whatever the last element of the list is. In our case "Clowns 101" gets removed.
2. .pop() is unique in that it will *return* the value that was removed. If we wanted to know what element was deleted, simply assign a variable to the call of the .pop() method. In this case, we assigned it to removed\_element.

Lastly let’s remove "Balloon Making":

cs\_topics.pop(2)  
print(cs\_topics)

Would output:

['Python', 'Data Structures', 'Algorithms']

Notice two things about this example:

1. The method can be called with an optional specific index to remove. In our case, the index 2 removes the value of "Balloon Making".
2. We don’t have to save the removed value to any variable if we don’t care to use it later.

***Note:*** Passing in an index that does not exist or calling .pop() on an empty list will both result in an IndexError.

**Consecutive Lists: Range**

Often, we want to create a list of consecutive numbers in our programs. For example, suppose we want a list containing the numbers 0 through 9:

my\_list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Typing out all of those numbers takes time and the more numbers we type, the more likely it is that we have a typo that can cause an error.

Python gives us an easy way of creating these types of lists using a built-in function called [range()](https://www.codecademy.com/resources/docs/python/built-in-functions/range?page_req=catalog).

The function range() takes a single input, and generates numbers starting at 0 and ending at the number **before** the input.

So, if we want the numbers from 0 through 9, we use range(10) because 10 is 1 greater than 9:

my\_range = range(10)  
print(my\_range)

Would output:

range(0, 10)

Notice something different? The range() function is unique in that it creates a *range object*. It is not a typical list like the ones we have been working with.

In order to use this object as a list, we have to first convert it using another built-in function called [list()](https://www.codecademy.com/resources/docs/python/built-in-functions/list?page_ref=catalog).

The list() function takes in a single input for the object you want to convert.

We use the list() function on our range object like this:

print(list(my\_range))

Would output:

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

**The Power of Range!**

By default, range() creates a list starting at 0. However, if we call range() with two inputs, we can create a list that starts at a different number.

For example, range(2, 9) would generate numbers starting at 2 and ending at 8 (just before 9):

my\_list = range(2, 9)  
print(list(my\_list))

Would output:

[2, 3, 4, 5, 6, 7, 8]

If we use a third input, we can create a list that “skips” numbers.

For example, range(2, 9, 2) will give us a list where each number is 2 greater than the previous number:

my\_range2 = range(2, 9, 2)  
print(list(my\_range2))

Would output:

[2, 4, 6, 8]

We can skip as many numbers as we want!

For example, we’ll start at 1 and skip in increments of 10 between each number until we get to 99 (one before 100):

my\_range3 = range(1, 100, 10)  
print(list(my\_range3))

Would output:

[1, 11, 21, 31, 41, 51, 61, 71, 81, 91]

Our list stops at 91 because the next number in the sequence would be 101, which is greater than or equal to 100 (our stopping point).

**Length**

Often, we’ll need to find the number of items in a list, usually called its *length*.

We can do this using a built-in function called [len()](https://www.codecademy.com/resources/docs/python/built-in-functions/len?page_ref=catalog).

When we apply len() to a list, we get the number of elements in that list:

my\_list = [1, 2, 3, 4, 5]  
   
print(len(my\_list))

Would output:

5

**Slicing Lists I**

In Python, often we want to extract only a portion of a list. Dividing a list in such a manner is referred to as *slicing*.

Lets assume we have a list of letters:

letters = ["a", "b", "c", "d", "e", "f", "g"]

Suppose we want to select from "b" through "f".

We can do this using the following syntax: letters[start:end], where:

* start is the index of the first element that we want to include in our selection. In this case, we want to start at "b", which has index 1.
* end is the index of *one more than* the last index that we want to include. The last element we want is "f", which has index 5, so end needs to be 6.

sliced\_list = letters[1:6]  
print(sliced\_list)

Would output:

["b", "c", "d", "e", "f"]

Notice that the element at index 6 (which is "g") is *not* included in our selection.

**Slicing Lists II**

Slicing syntax in Python is very flexible. Let’s look at a few more problems we can tackle with slicing.

Take the list fruits as our example:

fruits = ["apple", "cherry", "pineapple", "orange", "mango"]

If we want to select the *first n elements* of a list, we could use the following code:

fruits[:n]

For our fruits list, suppose we wanted to slice the first three elements.

The following code would start slicing from index 0 and up to index 3. Note that the fruit at index 3 (orange) is not included in the results.

print(fruits[:3])

Would output:

['apple', 'cherry', 'pineapple']

We can do something similar when we want to slice the *last n elements* in a list:

fruits[-n:]

For our fruits list, suppose we wanted to slice the last two elements.

This code slices from the element at index -2 up through the last index.

print(fruits[-2:])

Would output:

['orange', 'mango']

Negative indices can also accomplish taking *all but n last elements* of a list.

fruits[:-n]

For our fruits example, suppose we wanted to slice all but the last element from the list.

This example starts counting from the 0 index up to the element at index -1.

print(fruits[:-1])

Would output:

['apple', 'cherry', 'pineapple', 'orange'

**Counting in a List**

In Python, it is common to want to count occurrences of an item in a list.

Suppose we have a list called letters that represents the letters in the word “Mississippi”:

letters = ["m", "i", "s", "s", "i", "s", "s", "i", "p", "p", "i"]

If we want to know how many times i appears in this word, we can use the list method called [.count()](https://www.codecademy.com/resources/docs/python/lists/count?page_req=catalog):

num\_i = letters.count("i")  
print(num\_i)

Would output:

4

Notice that since .count() *returns* a value, we can assign it to a variable to use it.

We can even use .count() to count element appearances in a two-dimensional list.

Let’s use the list number\_collection as an example:

number\_collection = [[100, 200], [100, 200], [475, 29], [34, 34]]

If we wanted to know how often the sublist [100, 200] appears:

num\_pairs = number\_collection.count([100, 200])  
print(num\_pairs)

Would output:

2

**Sorting Lists I**

Often, we will want to sort a list in either numerical (1, 2, 3, …) or alphabetical (a, b, c, …) order.

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

Suppose that we have a list of names:

names = ["Xander", "Buffy", "Angel", "Willow", "Giles"]

Let’s see what happens when we apply .sort():

names.sort()  
print(names)

Would output:

['Angel', 'Buffy', 'Giles', 'Willow', 'Xander']

As we can see, the .sort() method sorted our list of names in alphabetical order.

.sort() also provides us the option to go in reverse. Instead of sorting in ascending order like we just saw, we can do so in descending order.

names.sort(reverse=True)  
print(names)

Would output:

['Xander', 'Willow', 'Giles', 'Buffy', 'Angel']

**Note:** The .sort() method does not return any value and thus does not need to be assigned to a variable since it modifies the list directly. If we do assign the result of the method, it would assign the value of None to the variable.

**Sorting Lists II**

A second way of sorting a list in Python is to use the built-in function sorted().

The [sorted()](https://www.codecademy.com/resources/docs/python/built-in-functions/sorted?page_req=catalog) function is different from the .sort() method in two ways:

1. It comes *before* a list, instead of after as all built-in functions do.
2. It generates a new list rather than modifying the one that already exists.

Let’s return to our list of names:

names = ["Xander", "Buffy", "Angel", "Willow", "Giles"]

Using sorted(), we can create a new list, called sorted\_names:

sorted\_names = sorted(names)  
print(sorted\_names)

This yields the list sorted alphabetically:

['Angel', 'Buffy', 'Giles', 'Willow', 'Xander']

Note that using sorted did not change names:

print(names)

Would output:

['Xander', 'Buffy', 'Angel', 'Willow', 'Giles']

**Review**

In this lesson, we learned how to:

* Add elements to a list by index using the .insert() method.
* Remove elements from a list by index using the .pop() method.
* Generate a list using the range() function.
* Get the length of a list using the len() function.
* Select portions of a list using slicing syntax.
* Count the number of times that an element appears in a list using the .count() method.
* Sort a list of items using either the .sort() method or sorted() function.

As you go through the exercises, feel free to use print() to see changes when not explicitly asked to do so.

# Combining Lists: The Zip Function

**Learn about a popular Python built-in function called zip().**

In Python, we have an assortment of built-in functions that allow us to build our programs faster and cleaner. One of those functions is **zip()**.

The **zip()** function allows us to quickly combine associated data-sets without needing to rely on multi-dimensional lists. While **zip()** can work with many different scenarios, we are going to explore only a single one in this article.

Let’s use a list of student names and associated heights as our example data set:

* Jenny is 61 inches tall
* Alexus is 70 inches tall
* Sam is 67 inches tall
* Grace is 64 inches tall

Suppose that we already had a list of names and a list of heights:

names = ["Jenny", "Alexus", "Sam", "Grace"]  
heights = [61, 70, 67, 64]

If we wanted to create a nested list that paired each name with a height, we could use the built-in function **zip()**.

The **zip()** function takes two (or more) lists as inputs and returns an object that contains a list of pairs. Each pair contains one element from each of the inputs. This is how we would do it for our **names** and **heights** lists:

names\_and\_heights = zip(names, heights)

If we were to then examine this new variable **names\_and\_heights**, we would find it looks a bit strange:

print(names\_and\_heights)

Would output:

<zip object at 0x7f1631e86b48>

This zip object contains the location of this variable in our computer’s memory. Don’t worry though, it is fairly simple to convert this object into a useable list by using the built-in function **list()**:

converted\_list = list(names\_and\_heights)  
print(converted\_list)

Outputs:

[('Jenny', 61), ('Alexus', 70), ('Sam', 67), ('Grace', 64)]

Notice two things:

1. Our data set has been converted from a zip memory object to an actual list (denoted by **[ ]**)
2. Our inner lists don’t use square brackets **[ ]** around the values. This is because they have been converted into [tuples](https://docs.python.org/3/tutorial/datastructures.html#tuples-and-sequences) (an immutable type of list).