Lists

You have earned 0 point(s) out of 0 point(s) thus far.

Introduction

There are various collection types in Python. While types such as int and str hold a single value, collection types hold multiple values.

In your programs, you usually need to group several items to render as a single object. We use collection types of data to do this job.

One of the most useful collections in Python is a list. In Python, a list is only an **ordered collection** of valid Python values.

The list type is probably the most commonly used collection type in Python. In spite of its name, a list is more like an array in some other languages (e.g. JavaScript).

### Creating a List

A list can be created by enclosing values, separated by commas, in square brackets 👉🏻[].

Let's create a simple list that includes some country names.



1

2

country = ['USA', 'Brasil', 'UK', 'Germany', 'Turkey', 'New

    Zealand']

That is our first list in this course. Now let's print the list.

input :



1

2

3

4

country = ['USA', 'Brasil', 'UK', 'Germany', 'Turkey', 'New

    Zealand']

print(country)

output :



1

2

['USA', 'Brasil', 'UK', 'Germany', 'Turkey', 'New Zealand']

https://drive.google.com/uc?export=view&id=1AAtBbH3F0uSt3rVDf7PhBS7L1VcMQAh-**Scratch Time ! :**Create this list with [**scratch**](https://scratch.mit.edu/projects/341599722/editor/).

**💡Tips:**

* All the country names are printed in the same order as they were stored in the list because lists are **ordered**.

Another way to create a list is to call the 'list()' function.

You do this when you want to create a list from an iterable object: that is, type of object whose elements you can import individually. The lists are iterable like other collections and string types. Let's create another list using list() function and compare with 👉🏻'[]'.

input :



1

2

3

4

5

6

7

8

string\_1 = 'I quit smoking'

new\_list\_1 = list(string\_1) # we created multi element

    list

print(new\_list\_1)

new\_list\_2 = [string\_1] # this is a single element list

print(new\_list\_2)

output :



1

2

3

['I', ' ', 'q', 'u', 'i', 't', ' ', 's', 'm', 'o', 'k', 'i'

    , 'n', 'g']

['I quit smoking']

**💡Tips:**

* Note that, using **list()** function, all characters of string\_1 including spaces was moved into a new\_list\_1.
* If you noticed, **lists** can contain **more than one** of the **same** value.

As it appears, the list() function creates a list that contains each component of a specific iterable object, such as a string. You can use square brackets or list() functions, depending on what you are going to do.

The components of a list are not limited to a single data type, given that Python is a dynamic language: e.g.

**mixed\_list = [11, 'Joseph', False, 3.14, None, [1, 2, 3]]**

**💡Tips:**

* As you see above, one or more of the **list elements** can even be a list.

Lists

You have earned 0 point(s) out of 0 point(s) thus far.

Basic Operations with Lists

In Python, there are many methods and functions for dealing with the list structures. You'll learn some of them which are basic and the most common. Let's begin:

In most cases, we'll have to make an empty list to fill it later with the data you want.



1

2

3

4

empty\_list\_1= []

empty\_list\_2 = list()

We can add an element into a list using .append() or .insert() methods.

* **.append()** : Append an object to end of a list. Using only list.append(element) syntax, returns none. If you want to see the new appended list, you have to call or print it. See the example :

input :



1

2

3

4

5

6

empty\_list\_1 = []

empty\_list\_1.append('114')

empty\_list\_1.append('plastic-free sea')

print(empty\_list\_1)

output :



1

2

['114', 'plastic-free sea']

input :



1

2

3

4

5

city = ['New York', 'London', 'Istanbul', 'Seoul',

    'Sydney']

city.append('Addis Ababa')

print(city)

output :



1

2

['New York', 'London', 'Istanbul', 'Seoul', 'Sydney',

    'Addis Ababa']

| **append** |
| --- |
| *Diagram of '.append( )' Method* |

looks like list.insert(index, object). See the example :

input :



1

2

3

4

5

city = ['New York', 'London', 'Istanbul', 'Seoul', 'Sydney'

    , 'Addis Ababa']

city.insert(2, 'Stockholm')

print(city)

output :



1

2

['New York', 'London', 'Stockholm', 'Istanbul', 'Seoul',

    'Sydney', 'Addis Ababa']

| **Tiobe_Index** |
| --- |
| *Diagram of '.insert( )' Method* |

We can remove the elements in lists using list.remove() method or sort the elements using list.sort() method. Examine the example :

input :



1

2

3

4

city = ['New York', 'London', 'Stockholm', 'Istanbul',

    'Seoul', 'Sydney', 'Addis Ababa']

city.remove('London')

print(city) # we have deleted 'London'

output :



1

2

['New York', 'Stockholm', 'Istanbul', 'Seoul', 'Sydney',

    'Addis Ababa']

input :



1

2

3

4

city = ['New York', 'Stockholm', 'Istanbul', 'Seoul',

    'Sydney', 'Addis Ababa']

city.sort() # lists the items in alphabetical order

print(city)

output :



1

2

['Addis Ababa', 'Istanbul', 'New York', 'Seoul',

    'Stockholm', 'Sydney']

**💡Tips:**

* Remember! Elements of a list are counted from left to right and start with zero as in string types.

Likewise, the length of the list elements can be calculated with the len() function also. Let's calculate the length of 'city' variable we have.

input :



1

2

3

city = ['Addis Ababa', 'Istanbul', 'New York', 'Seoul',

    'Stockholm', 'Sydney']

print(len(city))

output :



1

2

6

**✏️Homework:**

* Guess and figure out the output of this syntax :



1

2

my\_list = [1, 3, 5, 7]

print(my\_list \* 3)

Show the Answer

One of the important operations of the lists is assigning an element to the specific index number.

input :



1

2

3

4

city = ['New York', 'Stockholm', 'Istanbul', 'Seoul',

    'Sydney', 'Addis Ababa']

city[1] = 'Melbourne' # we assign 'Melbourne' to index 1

print(city)

output :



1

2

['New York', 'Melbourne', 'Istanbul', 'Seoul', 'Sydney',

    'Addis Ababa']

**✏️Homework:**

* Examine the use of index(), del() and pop() functions.

There are many other 'list operations' (mutable sequence types operations) methods [**here**](https://docs.python.org/3/library/stdtypes.html#mutable-sequence-types) that you can examine in detail.

## Accessing Lists

### Introduction

You know that there are several types of collections for storing data in Python, like **list, tuple, dictionary**.

Each item or element in a list, as well as every character in a string, has an index corresponding to their location. Using indexes, we can access elements within a sequence. Now, let's see how can we do that?

Accessing Lists

You have earned 0 point(s) out of 0 point(s) thus far.

Indexing a List

If we want to access or use the elements of a list, we can do that using index numbers of the list enclosed by **square brackets**.

**⚠️Avoid ! :**

* Do not start indexing with **1**. The first index of the element of a list is **0**. We will never stop remembering that!

First, let's begin with a simple example :

input :



1

2

3

4

colors = ['red', 'purple', 'blue', 'yellow', 'green']

print(colors[2]) # If we start at zero,

# the second element will be 'blue'.

output :



1

2

blue

Now, let's learn the subject in detail through the examples :

input :



1

2

3

4

5

6

7

city = ['New York', 'London', 'Istanbul', 'Seoul',

    'Sydney']

city\_list = []

city\_list.append(city) # we have created a nested list

print(city\_list)

output :



1

2

[['New York', 'London', 'Istanbul', 'Seoul', 'Sydney']]

**city\_list** includes only **one** element which is the city list.

**💡Tips :**

* If you notice that city\_list has double square brackets.

input :



1

2

3

city\_list = [['New York', 'London', 'Istanbul', 'Seoul',

    'Sydney']]

print(city\_list[0]) # access to first and only element

output :



1

2

['New York', 'London', 'Istanbul', 'Seoul', 'Sydney']

'**city\_list[0]**' is a list type data. So that, we can still access its elements via indexing. Let's access its second element :

input :



1

2

3

city\_list = [['New York', 'London', 'Istanbul', 'Seoul',

    'Sydney']]

print(city\_list[0][2])

output :



1

2

Istanbul

'**city\_list[0][2]**' is a string type data. So, we can also access its elements via indexing. Let's access its third element :

input :



1

2

3

city\_list = [['New York', 'London', 'Istanbul', 'Seoul',

    'Sydney']]

print(city\_list[0][2][3])

output :



1

2

a

Accessing Lists

You have earned 0 point(s) out of 0 point(s) thus far.

Slicing a List

We can access individual elements of a list, as well as part of those items. We use index numbers again for slicing but we do it by typing it a little differently. Look at the example :

input :



1

2

3

numbers = [1, 3, 5, 7, 9, 11, 13, 15, 17]

print(numbers[2:5]) # we get the elements from index=2 to

    index=5(5 is not included)

output :



1

2

[5, 7, 9]

**💡Tips :**

* Slicing is just similar to indexing. The difference is adding **colon** or **colons** in square brackets.

In slicing, pay attention to the stop index in square brackets. We got the elements from **index=2** to **index=5** (5 is not included): It means that we got 'second', 'third' and 'forth' element of the list.

You can keep in mind the formula syntax below for slicing a sequence. From '**start**' to '**stop-1**', by '**step**'.

**The formula syntax is : sequence[start:stop:step]**

This formula produces a slice of the sequence where **start** is an index of the first element required (the element is included in the slice) and **stop** is an index of the end element (the element is not included in the slice), **step** is an interval between elements to be chosen.

Now let's apply this formula on a few examples. In this example, we will create a list of numbers from 1 to 10 using 'range()' function and select even ones:

input :



1

2

3

4

5

count = list(range(11))

print(count)

print(count[0:11:2])

output :



1

2

3

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

[0, 2, 4, 6, 8, 10]

By the way, range() function returns an object that produces a sequence of integers from start (including) to stop (excluding) by step.

**The formula syntax is : range(start, stop[, step])**

Each part of the slice has a default value, so they are **optional**. If we don't assign a value to the **start** index, it is considered to be **0**; if we don't assign a value to the **stop** index, it will be the **same as** the **length** of the sequence.

* **my\_list[:]**: returns the full copy of the sequence
* **my\_list[start:]** : returns elements from start to the end element
* **my\_list[:stop]** : returns element from the 1st element to stop-1
* **my\_list[::step]** : returns each element with a given step

Let's do some more examples to grasp it.

The following example outputs the same as the input list (animals).

input :



animals = ['elephant', 'bear', 'fox', 'wolf', 'rabbit',

    'deer', 'giraffe']

print(animals[:]) # all elements of the list

output :



['elephant', 'bear', 'fox', 'wolf', 'rabbit', 'deer',

    'giraffe']

The following example slices the animals starts at **index=3** to the end.

input :



animals = ['elephant', 'bear', 'fox', 'wolf', 'rabbit',

    'deer', 'giraffe']

print(animals[3:])

output :



['wolf', 'rabbit', 'deer', 'giraffe']

The following example slices the animals starts at **index=0** to the **index=4**.

input :



animals = ['elephant', 'bear', 'fox', 'wolf', 'rabbit',

    'deer', 'giraffe']

print(animals[:5])

output :



['elephant', 'bear', 'fox', 'wolf', 'rabbit']

And the last example slices animals starts at **index=0** to the **end** with **2 step**.

input :



animals = ['elephant', 'bear', 'fox', 'wolf', 'rabbit',

    'deer', 'giraffe']

print(animals[::2])

output :



['elephant', 'fox', 'rabbit', 'giraffe']

Q: In Python what is slicing?  
A: A mechanism to select a range of items from sequence types like list, tuple, strings etc. is known as slicing.

Accessing Lists

You have earned 0 point(s) out of 0 point(s) thus far.

Negative Indexing & Slicing

**Negative indexing** is the best and shortest way to reach the elements at the end of the list. The negative indexing works in reverse. We can reach the last element of a list as list\_name[-1]. See the example below :

input :



city = ['New York', 'London', 'Istanbul', 'Seoul',

    'Sydney']

print(city[-4])

output :



London

| **negative_index** |
| --- |
| *Diagram of Negative Indexing* |

**Negative slicing** also works similarly, as we see in single element access. In this case, **step index** can also be negative. If the step index is negative the elements of sequence will return in **reverse order**. Let's see in examples :

input :



reef = ['swordfish', 'shark', 'whale', 'jellyfish',

    'lobster', 'squid', 'octopus']

print(reef[-3:])

output :



['lobster', 'squid', 'octopus']

| **negative_index_1** |
| --- |
| *Diagram-1 of Negative Slicing* |

input :



reef = ['swordfish', 'shark', 'whale', 'jellyfish',

    'lobster', 'squid', 'octopus']

print(reef[:-3])

output :



1

2

['swordfish', 'shark', 'whale', 'jellyfish']

| **negative_index_2** |
| --- |
| *Diagram-2 of Negative Slicing* |

input :



1

2

3

reef = ['swordfish', 'shark', 'whale', 'jellyfish',

    'lobster', 'squid', 'octopus']

print(reef[::-1]) # we have produced the reverse of the

    list

output :



1

2

['octopus', 'squid', 'lobster', 'jellyfish', 'whale',

    'shark', 'swordfish']

| **negative_index_3** |
| --- |
| *Diagram-3 of Negative Slicing* |

input :



1

2

3

reef = ['swordfish', 'shark', 'whale', 'jellyfish',

    'lobster', 'squid', 'octopus']

print(reef[::-2])

output :



1

2

['octopus', 'lobster', 'whale', 'swordfish']

**💡Tips :**

* If you choose negative step with the start and end indexes together, those should be used accordingly, that is, the end index should be less than the start index.

input :



1

2

3

4

5

odd\_no = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

print(odd\_no[7:3:-1])

print(odd\_no[2:6:-1])

output :



1

2

3

[8, 7, 6, 5]

[]

If you eager to find **more on lists** see [**here**](https://docs.python.org/3.8/tutorial/datastructures.html#more-on-lists).

**Q**: What does list[::-1] do?  
**A**: list[::-1] is used to reverse the order of a sequence of the elements in the list.

**- Interview Q&A**

Tuples

You have earned 0 point(s) out of 0 point(s) thus far.

Definitions

Up to this section of our lesson, we saw the most used collection types of Python : list.

A tuple is another collection type that can hold multiple data very similar to the list.

The most important difference from the list is that the tuple is **immutable**.

Therefore, methods like append() or remove() do not exist in the operations of this type.

Tuples are commonly used for small collections of values that will not need to change, such as an IP address and port. If we have unchanged data, we should choose **tuples** because it is much **faster than** **lists**.

We used square brackets 👉🏻'[]' to define the lists. In the tuple, normal parentheses 👉🏻'()' are used.

The same indexing rules for lists also apply to tuples. Tuples can also be nested and the values can be any valid Python valid.

Q: What is the difference between list and tuple?  
A:  
LISTs :

* Lists are mutable i.e they can be edited.
* Lists are slower than tuples.
* Syntax: list\_1 = [True, ‘Space’, 20]

TUPLEs :

* Tuples are immutable (tuples are lists which can’t be edited).
* Tuples are faster than list.
* Syntax: tup\_1 = (True, ‘Space’ , 20)

Tuples

You have earned 0 point(s) out of 0 point(s) thus far.

Creating a Tuple

A tuple also can be created by enclosing values, separated by commas, in **parentheses**.

You can compare tuple to a case. When you put the data that you want it to not change and close the lid, you can no longer change this data, modify its size and edit it.

Let's create a simple empty tuple :



empty\_tuple = ()

This is our first tuple in this course. Now let's print its type.

input :



empty\_tuple = ()

print(type(empty\_tuple))

output :



<class 'tuple'>

If you want to create a single element tuple, you should use a comma.

input :



try\_tuple = ('love')

print(try\_tuple)

print(type(try\_tuple)) # it's not tuple type.

output :



love

<class 'str'>

It occurs in only single element tuples and we can fix the problem using **comma** at the end of the element.

**💡Tips:**

* Remember to always use a comma when defining a singleton tuple.

input :



try\_tuple = ('love',)

print(try\_tuple)

print(type(try\_tuple)) # it's a tuple type.

output :



('love',)

<class 'tuple'>

Actually, if your tuple contains more than one element, separating elements with commas will be enough.

Another way to create a tuple is to call the tuple() function. You do this when you want to create a tuple from an iterable object: that is, a type of object whose elements you can import individually.

The tuple is also iterable like other collections and string types. Let's create another tuple using tuple() function. With this function, you can create an empty tuple as well.

Let's examine some examples of creating tuples :

input :



planets = 'mercury', 'jupiter', 'saturn'

print(planets)

print(type(planets))

output :



('mercury', 'jupiter', 'saturn')

<class 'tuple'>

input :



empty\_tuple\_1 = tuple()

print(empty\_tuple\_1)

print(type(empty\_tuple\_1))

output :



()

<class 'tuple'>

It is easy to convert between list and tuple as in the examples below :

input :

1

my\_tuple=(1, 4, 3, 4, 5, 6, 7, 4)

my\_list = list(my\_tuple)

print(type(my\_list), my\_list)

output :



<class 'list'> [1, 4, 3, 4, 5, 6, 7, 4]

input :



my\_list = [1, 4, 3, 4, 5, 6, 7, 4]

my\_tuple = tuple(my\_list)

print(type(my\_tuple), my\_tuple)

output :



<class 'tuple'> (1, 4, 3, 4, 5, 6, 7, 4)

An iterable string can be converted to a tuple :

input :



mountain = tuple('Alps')

print(mountain)

output :



('A', 'l', 'p', 's')

Tuples

You have earned 0 point(s) out of 0 point(s) thus far.

How can We Use a Tuple ?

If you want, let's take a look at the common features of the list and tuple. So you can have an idea of what to do with tuples.

Both lists and tuples are ordered. It means that when storing elements to these containers, you can sure that their order will remain the same. You can also duplicate values or mix different data types in tuples.

input :



mix\_value\_tuple = (0, 'bird', 3.14, True)

print(len(mix\_value\_tuple))

output :



4

As we stated at the beginning, just like lists, tuples support indexing :

input :



even\_no = (0, 2, 4)

print(even\_no[0])

print(even\_no[1])

print(even\_no[2])

print(even\_no[3])

output :



0

2

4

-----------------------------------------------------------

    ----------------

print(even\_no[3]) : IndexError: tuple index out of range

And one of the most important differences of tuples from lists is that 'tuple' object does not support **item assignment**. Yes, because tuple is immutable. See the example :

input :



city\_list = ['Tokyo', 'Istanbul', 'Moskow', 'Dublin']

city\_list[0] = 'Athens'

city\_list[1] = 'Cairo'

print(city\_list)

output :



['Athens', 'Cairo', 'Moskow', 'Dublin']

input :



city\_list = ['Tokyo', 'Istanbul', 'Moskow', 'Dublin']

city\_tuple = tuple(city\_list)

city\_tuple[0] = 'New York' # you can't assign a value

output :



1

2

3

-----------------------------------------------------------

    ----------------

TypeError: 'tuple' object does not support item assignment

**✏️Homework:**

* Examine the use of .index() and .count() methods of the tuples.

You have earned 0 point(s) out of 0 point(s) thus far.

Benefits of Immutability

Let's take a look at the basic advantages of tuples :

* Tuples are faster and more powerful in-memory than lists. You should give it a thought whenever you need to deal with large amounts of data. If you don't want to change your data you may have to choose tuples.
* Because of its immutability, the data stored in a tuple can not be altered by mistake.
* A tuple can be used as a dictionary (we will see in the next lesson) key, while 'TypeError' can result in lists as keys. And this is the usefulness of tuples in the data processing.

If you want to go deep into tuples, you can find lots of information [**here**](https://docs.python.org/3.8/tutorial/datastructures.html#tuples-and-sequences).

Unfold (unco lapse) all regions

Toggle breakpoint