What is the difference between list and tuple:: <https://www.afternerd.com/blog/difference-between-list-tuple/> (well explained)

In Python, strings and lists are both examples of sequences.

There are other sequences too, and they all share a bunch of operations like

* iterating over them using for-loops
* indexing using the len function to know the length of the sequence
* using plus to concatenate two sequences and
* using in to verify if the sequence contains an element.

**List**

>>x=["Now","we","are","cooking!"]

>>type(x)

<class 'list'>

>>>len(x)

4

>>>"we" in x

True

>>>"today" in x

False

The result of this check is a Boolean, which we can use as a condition for branching or looping. We can also use indexing to access individual elements depending on their position in the list. Remember that the first element is given the index zero. This means the last index of the list will be the length of the list minus one

>>>print(x[0])

Now

>>>print(x[3])

cooking!

As with strings, we can also use indexes to create a slice of the list. For this, we use ranges of two numbers separated by a colon. Again, the second element isn't included in the slice. So the range goes to the second index minus one.

>>>x[1:3]

['we', 'are']

One of the ways that lists and strings are different is that lists are mutable. Which is another fancy word to say that they can change. This means we can add, remove, or modify elements in a list.

All List-specific operations and methods([https://docs.python.org/3/library/stdtypes.html#mutable-sequence-types](https://docs.python.org/3/library/stdtypes.html" \l "mutable-sequence-types)) 🡸 Everything you need is included

Common sequence operations between List and tuples (<https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range>)

Insert, remove, pop method

list=["ifthekher","shoib","zidan","jobayed"]  
list.insert(0,"touha")  
print(list)

output:: ['touha', 'ifthekher', 'shoib', 'zidan', 'jobayed']

What happens if we use a number larger than the length of the list?

list=["ifthekher","shoib","zidan","jobayed"]  
list.insert(7,"ashraf")  
print(list)

['ifthekher', 'shoib', 'zidan', 'jobayed', 'ashraf']

If we use an index higher than the current length, the element just gets added to the end. You can pass any number to insert but usually, you either add at the beginning using insert at the zero index or at the end using append

list=["ifthekher","shoib","zidan","jobayed","ifthekher"]  
list.remove("ifthekher")  
print(list)

['shoib', 'zidan', 'jobayed', 'ifthekher']

The remove method removes from the list the first occurrence of the element we pass to it.

What happens if the element is not in the list? We get an ValueError.

Another way we can remove elements is by using the pop method, which receives an index. The pop method returns the element that was removed at the index that was passed.

list=["ifthekher","shoib","zidan","jobayed","ifthekher"]  
list.pop(0)  
print(list)

['shoib', 'zidan', 'jobayed', 'ifthekher']

You can remove elements from the list using the **remove** method. This method takes an element as a parameter, and removes the first occurrence of the element. If the element isn’t found in the list, you’ll get a **ValueError** error explaining that the element was not found in the list.

You can also remove elements from a list using the **pop** method. This method differs from the remove method in that it takes an index as a parameter, and returns the element that was removed. This can be useful if you don't know what the value is, but you know where it’s located. This can also be useful when you need to access the data and also want to remove it from the list.

Finally, you can change an element in a list by using indexing to overwrite the value stored at the specified index. For example, you can enter **list[0] = "Old data"** to overwrite the first element in a list with the new string "Old data".

**Tuples**

There are a number of data types in Python that are all sequences.

* Strings are sequences of characters and are immutable.
* Lists are sequences of elements of any type and are mutable.
* A third data type that's a sequence and also closely related to lists is the tuple. Tuples are sequences of elements of any type that are immutable. We write tuples in parentheses instead of square brackets.
* TUPLE=("ifthekher","123","jobayed","321")

Why do we even need another sequence type? Weren't lists great? Yes, lists are great. They can hold any number of elements and we can add, remove and modify their contents as much as we want, but there are cases when we want to make sure an element in a certain position or index refers to one specific thing and won't change. In these situations, lists won't help us.

Tuples are used for lots of different things in Python. One common example is the return value of functions. When a function returns more than one value, it's actually returning a tuple.

def tuple\_check(n):  
 n1=n\*1  
 n2=n\*2  
 n3=n\*3  
 return n1,n2,n3  
print(type(tuple\_check(2)))

print(tuple\_check(2))

<class 'tuple'>

(2, 4, 6)

One interesting thing we can do with tuples is unpack them. This means that we can turn a tuple of three elements into three separate variables. Because the order won't change, we know what those variables are present

def tuple\_check(n):  
 n1=n\*1  
 n2=n\*2  
 n3=n\*3  
 return n1,n2,n3  
n1,n2,n3=tuple\_check(2)  
print(n1,n2,n3)

2 4 6

In Python, it's really common to use tuples to represent data that has more than one value and that needs to be kept together. For example, you could use a tuple to have a filename and it's size, or you could store the name and email address of a person, or a date and time and the general health of the system at any point in time.

Say you have a list of tuples containing two strings each. The first string is an email address and the second is the full name of the person with that email address. You want to write a function that creates a new list containing one string per person including their name and the email address between angled brackets. the format usually used in emails like this “name<[name@example.com](mailto:name@example.com)>”.

def full\_emails(people):  
 rearranged=[]  
 for email,name in people:  
 rearranged.append("{name} <{email}>".format(name=name,email=email))  
 print(rearranged)  
  
people=[("ifthekher239@gmail.com","ifthekher"),("raz1971bangalee@gmail.com","raz bangalee")]  
full\_emails(people)

output::

['ifthekher <ifthekher239@gmail.com>', 'raz bangalee <raz1971bangalee@gmail.com>']

N:B:: See at the line where for loop started, see how entries of a tuple are unpacked and used in the loop.

Enumerate() in Python

The enumerate() function takes a list as a parameter and returns a tuple for each element in the list. The first value of the tuple is the index and the second value is the element itself.

A lot of times when dealing with iterators, we also get a need to keep a count of iterations. Python eases the programmers’ task by providing a built-in function enumerate() for this task.  
Enumerate() method adds a counter to an iterable and returns it in a form of enumerate object. This enumerate object can then be used directly in for loops or be converted into a list of tuples using list() method.

**Syntax:**

enumerate(iterable, start=0)

**Parameters:**

**Iterable:** any object that supports iteration

**Start:** the index value from which the counter is to be started, by default it is 0

# Python program to illustrate enumerate function

l1 = ["eat","sleep","repeat"]

s1 = "geek"

# creating enumerate objects

obj1 = enumerate(l1)

obj2 = enumerate(s1)

print "Return type:",type(obj1)

print list(enumerate(l1))

# changing start index to 2 from 0

print list(enumerate(s1,2))

Return type: < type 'enumerate' >

[(0, 'eat'), (1, 'sleep'), (2, 'repeat')]

[(2, 'g'), (3, 'e'), (4, 'e'), (5, 'k')]

**Using Enumerate object in loops**

# Python program to illustrate enumerate function in loops

L1 = ["eat","sleep","repeat"]

# printing the tuples in object directly

for ele in enumerate(L1):

print (ele)

print()

# changing index and printing separately

for count,ele in enumerate(L1,100):

print (count,ele)

(0, 'eat')

(1, 'sleep')

(2, 'repeat')

100 eat

101 sleep

102 repeat

Because we use the range function so much with for loops, you might be tempted to use it for iterating over indexes of a list and then to access the elements through indexing. You could be particularly inclined to do this if you're used to other programming languages before. Because in some languages, the only way to access an element of a list is by using indexes. Real talk, this works but looks ugly. It's more idiomatic in Python to iterate through the elements of the list directly or using enumerate when you need the indexes like we've done so far. There are some specific cases that do require us to iterate over the indexes, for example, when we're trying to modify the elements of the list we're iterating. By the way, if you're iterating through a list and you want to modify it at the same time, you need to be very careful. If you remove elements from the list while iterating, you're likely to end up with an unexpected result. In this case, it might be better to use a copy of the list instead.

**List Comprehensions::**

List comprehensions let us create new lists based on sequences or ranges.

multiples=[]  
for x in range(1,11):  
 multiples.append(x\*7)  
  
print(multiples)

Or you can do the very same task by list comprehensions, like this way

multiples=[ x\*7 for x in range(1,11)]  
  
print(multiples)

So we can use this technique whenever we want to create a list based on a range like in this example. Or based on the contents of a list a tuple a string or any other Python sequence.

languages=["python","java","perl","C","Go"]  
lengths=[len(language) for language in languages]  
print(lengths)

List comprehensions also let us use a conditional clause.

list=[x for x in range(1,10) if x%2 == 0]  
print(list)

### **List comprehension**

* [expression for variable in sequence] Creates a new list based on the given sequence. Each element is the result of the given expression.
* [expression for variable in sequence if condition] Creates a new list based on the given sequence. Each element is the result of the given expression; elements only get added if the condition is true.

WARNING::

List comprehensions can be really powerful, but they can also be super complex, resulting in code that’s hard to read. Be careful when using them, since it might make it more difficult for someone else looking at your code to easily understand what the code is doing.