Chapter 5 - Lists & Tuples

5 Lists and Tuples

In this chapter, we will be combining 2 very similar data structures: Lists and Tuples. Both of these data structures share many similar operators and iterating techniques. However, we will start to focus more on functions for lists after their similarities end as Lists are mutable and Tuples are immutable.

Both lists and tuples are iterable objects as each **element** in them is assigned a number for its position (we also call this the **index**).

5.1 Concept of Indexing and Slicing

What is *Indexing*? Indexing means to refer to a specific element of an iterable object. Iterable objects in Python, are objects that are capable of returning its members one at a time using a loop. Indexing is done via the square brackets `[]`. So how do we know the indexes of an iterable object?

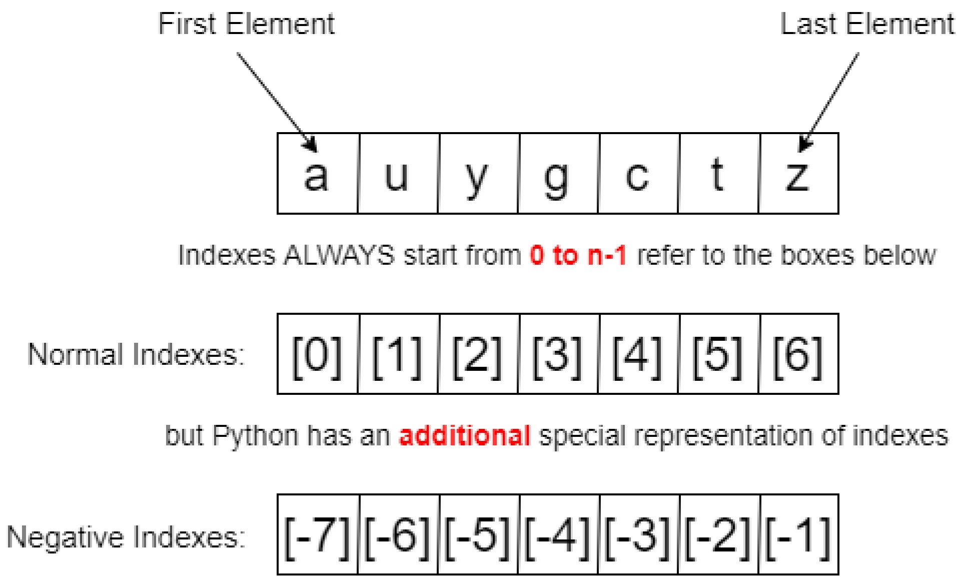

Figure 1: Reading indexes of Iterable Objects.

Referring to figure 1 above, the rule for iterable objects in most programming languages is that the **first** element always starts with the index **0** all the way to **n-1** (where `n` is the size of the iterable object) which will be the last element of the iterable object. However, Python has an additional method of accessing indexes of iterable objects in the reversed direction (ie from right to left). It uses the negative number notation, so it starts from the **last** element at **-1** to the first element at **-n**.

Slicing is done using the colon (`:`) operator where it means to access elements from a start to an end index. Note that if either the range's start or end index is omitted, it will get the range of indexes from the given index to end of the "last element" in either left or right direction. Slicing operation can also have step values like the `range()` function that we have seen in the chapter on *Loops*.

5.2 Creation

There are several methods to create Lists and Tuples but we will look at 2 basic ways and an advanced method for lists in section 5.13.

```
# method 1: using its object name
list([8,5,2,5])
tuple([5,2,1,7])
# note the difference for single tuple creatation
# without the comma, it is regarded as a normal string
tuple(['foo',])

# method 2: using directly using the brackets notation
list_var = [1,2,3,6]     # list
tuple_var = (1,2,3,6)    # tuple
```

Lists and tuples are also both able to store different types of elements in them like so

```
# Lists
list1 = ["abdc", 1, Employee]   # string, number, class instance object
list2 = [1,2,3,5]               # number
list3 = ["a", "b", "c"]         # string

# Tuples
tuple1 = ("abdc", 1, Employee)
tuple2 = (1,2,3,5)
tuple3 = ("a", "b", "c")
```

5.3 Modifying

If we tried to modify a element in a tuple, it would result in an error like so

```
list_var = [1,2,3,6]     # list
tuple_var = (1,2,3,6)    # tuple

# attempting to change the some of the data
# in both variables
list_var[1] = 10
print("list_var updated to ", list_var)
tuple_var[1] = 10
print("tuple_var updated to ", tuple_var)
```

the output is

```
list_var updated to  [1, 10, 3, 6]
Traceback (most recent call last):
  File "main.py", line 6, in <module>
    tuple_var[1] = 10
TypeError: 'tuple' object does not support item assignment
```

However, it is possible to combine 2 tuples or lists together using the plus (`+`) operator or the list `extend()` function. Note that when the to tuples or lists are combined together, the second tuple or list being added is always added to the end of the first list.

```
list_var01 = [1,2,3,6]
list_var02 = ['foo', 'bar']
tuple_var01 = (1,2,3,6)
tuple_var02 = ('red', 'fox')

# combining 2 tuples and storing the result in tuple_var01
tuple_var01 += tuple_var02

# combining 2 lists and storing the result in list_var01
# method 1: plus operator
list_var01 += list_var02
# method 2: extend() function
list_var01.extend(list_var02)
```

5.4 Duplication

The duplication of a list or tuple is done using the multiply (`*`) operator but bear in mind that all the elements will be duplicated as well.

```
['bar', 8] * 3
# output: ['bar', 8, 'bar', 8, 'bar', 8]
('red', 4) * 3
# output: ('red', 4, 'red', 4, 'red', 4)
```

5.5 Comparison/Membership

Lists and tuples comparison are done using the comparison operators. It works by comparing each element at the same index of both lists or tuples. Note that for the comparison to work, the lists and tuples that are to be compared have store the same type of data.

```
list_var01 = [1,2,3,6]
list_var02 = [1,2,3,6]
tuple_var01 = (1,2,3,6)
tuple_var02 = (1,2)

list_var01 == list_var02  # output: True
tuple_var01 > tuple_var02  # output: False
```

To check if a particular element is in a list or tuple, we can use the `in` membership operator. This will return either a `True` or `False` value.

```
list_var01 = ['bird', 'croc', 'elephant']
tuple_var01 = ('foo', 'bar', 'werewolf')

'bluejay' in list_var01   # output: False
'foo' in tuple_var01   # output: True
```

5.6 Counting

Counting in the number of occurrences of an element in a list or tuple is done with the `count()` function.

```python
list_var01 = ['bird', 'croc', 'elephant', 'bird', 'croc', 'croc']
tuple_var01 = ('foo', 'bar', 'werewolf', 555, 555, 789)

list_var01.count('bird')  # output: 2
tuple_var01.count('foo')  # output: 1
```

5.7 Iterating

Iterating thought lists and tuples can be done using both `for` and `while` loops but generally a `for` loop is used most often. There are 2 ways to iterate a list or tuple, using the normal `for..in` or using the indexing method. Example

```python
list_var01 = [1,2,3,6]
tuple_var01 = (1,2,3,6)
# method 1: normal for..in
for elem in list_var01:
    print(elem)

# method 2: indexing method
for idx in range(len(tuple_var01)):
    print(tuple_var01[idx])
```

5.8 Indexing

Let's we know the elements in a list or tuple but we do not know the index of the element. We can use the `index()` function to find its index. Note that the `index()` function only returns the first occurrence of the element in the list or tuple.

```python
list1 = [1,2,3,4,5]     # list
tuple1 = (100,99,98,97)    # tuple

list1.index(4)  # output: 3
tuple1.index(98)  # output: 2
```

5.9 Slicing

As we have seen from earlier examples, elements in the lists and tuples can be accessed through their indexes placed within the square brackets `[]`. And we have learnt from section 5.1 that slicing can be done on iterable objects. This rule also applies to both Lists and Tuples.

```python
list1 = [1,2,3,4,5]      # list
tuple1 = (100,99,98,97)     # tuple

print("list1 element from range 1 to 4: ", list1[1:4])
print("list1 element from range -2 to end (right direction): ", list1[-2:])
print("getting every second element of list1: ", list1[::2])
print("reversing list1: ", list1[::-1])
print()
print("tuple1 element from range 1 to 4: ", tuple1[1:4])
print("tuple1 element from range -2 to end (left direction): ", tuple1[:-2])
print("getting every second element of tuple1: ", tuple1[::2])
print("reversing tuple1: ", tuple1[::-1])
```

the output is

```
list1 element from range 1 to 4:  [2, 3, 4]
list1 element from range -2 to end (right direction):  [4, 5]
getting every second element of list1:  [1, 3, 5]
reversing list1:  [5, 4, 3, 2, 1]

tuple1 element from range 1 to 4:  (99, 98, 97)
tuple1 element from range -2 to end (left direction):  (100, 99)
getting every second element of tuple1:  (100, 98)
reversing tuple1:  (97, 98, 99, 100)
```

One thing to note about slicing is that it will **always** return a new object of the same type which you can then use to replace the existing variable if necessary. Slicing can also be used to manipulate a range of elements in a list.

```python
list1 = [1,2,3,4,5]
# the right side element can be any Python iterable object
list1[1:3] = 'possible'
print(list1)
# output: [1, 'p', 'o', 's', 's', 'i', 'b', 'l', 'e', 4, 5]
```

However there is a list function that reverses the elements in place, that means there is no new list returned and this function is called `reversed()`.

```python
list1 = [1,2,3,4,5]
list1.reverse()  # nothing is returned
print(list1)  # output: [5, 4, 3, 2, 1]
```

5.10 Sorting

Sorting elements in a list or tuple can be done using 2 functions, `sort()` from the list functions or `sorted()` from the Python built-in functions. The difference between the 2 functions is that `sort()` is built for lists only and does the sorting in place and `sorted()` can be used for any Python iterable object and always returns a `list` datatype copy of the sorted object. Both functions are able to accept custom functions that determines how the elements are to be sort if we do not want to follow the default sorting behaviour (we will learn more about this in the chapter for Dictionaries). The default sorting behaviour is to sort via ascending order and this can be changed via the `reversed` argument.

```
list1 = [8,9,14,56,247,96,5]
# list function, ascending order, in place
list1.sort()

# built-in Python function, descending order
sorted_list = sorted(list1, reverse=True)
```

5.11 Adding Items to Lists

In the section *Modifying*, we have seen that we can use the plus (`+`) operator to add 2 lists but what if we only want to add 1 element and not a list of elements? For this, we use the `append()` function from the list object and if we require to add this element at an specific index, the `insert(<index>, <element>)` function is used.

```
list1 = ['one', 'two', 'three']

# adding element to the end of the list
list1.append('five')
print(list1)
# output: ['one', 'two', 'three', 'five']

# insert element at index 3 of the list
list1.insert(3, 'four')
print(list1)
# output: ['one', 'two', 'three', 'four', 'five']
```

5.12 Deleting/Removing Elements from Lists

Deleting or removing element/s from the list can be done using several ways. Most delete or removal functions/statement does their operations in place. However it is imperative to remove elements from lists using a loop as each time an element is deleted or removed, Python will resize the list and the element's indexes will change.

- using the using the build-in `del` statement (in place).

```python
list1 = [5,9,3,9,3,4,0]
# deleting element at index 3
del list1[3]

# mixing the del statement with the slice operation
# removing every element with an odd index
list1 = [5,9,3,9,3,4,0]
odd = 2
del list1[odd-1::odd]
```

- using the `remove()` function (in place). Note that the `remove()` function will only remove the first occurrence of the element in the list.

```python
list1 = [5,9,3,9,3,4,0]
# remove the first occurrence of the element 3
list1.remove(3)
```

- using the `pop()` function (returns the element). This function will remove and return the removed element. By default, it will remove the last element from the list unless an index is provided.

```python
list1 = [5,9,3,9,3,4,0]
# remove and return element at index 3
list1.pop(3)

# remove and return the last element of the list
list1.pop()
```

- to remove all elements of the list, we can either use the `del` statement or the list `clear()` function.

```python
# clearing all elements in a list
# using del statement
del list1[:]

# using list clear function
list1.clear()
```

5.13 List Comprehension

List comprehension is another method for creating lists that combines a `for` loop and a conditional within square brackets (`[]`). The format of its syntax is as follows:

```python
new_list = [expression for m in iterable if condition]
# or
new_list = [expression if condition else false_expression for m in iterable]
```

The `if` conditional is optional but the `for` loop is mandatory. Let's break down the parts:

- `expression` is the value of `m` or a call to a functions or any other valid expression that returns a value.

- `m` is an object or value of the iterable object
- `iterable` - an object that can be a list, set, function, generator or any other object that can return its elements one at a time.
- `condition` - the condition that checks the value of `m` before passing it to the relevant `expression`.
- `false_expression` - the expression that gets evaluated in the event that the `if` condition results in a `False` value.

> Note that **all** list comprehensions are single line statements therefore you have to be mindful of the statement's character length and complexity.

The above syntax is also equivalent to the following:

```python
new_list = []
for m in iterable:
    if condition:
        new_list.append(m)
    else:
        new_list.append(m)
```

The main benefit of using list comprehensions is that it is a single tool that combines list creation with mapping and/or filtering functionalities. This is also a Pythonic way to create a list. An example would be, if you would like to a 10% discount to all item prices which are above $50 and give the item prices less than $50 a 5% discount.

```python
original_prices = [80.25, 11.45, 104.22, 86.78, 45.92, 51.16]
prices = [round((i-i*0.1),2) if i > 50 else round((i-i*0.05), 2) for i in original_prices]
# [72.22, 10.88, 93.8, 78.1, 43.62, 46.04]
```

Although list comprehensions are Pythonic, it is not applicable in certain situations. There are nested comprehensions used in combination with lists and other iterable objects such as dictionary and sets to create a combination objects or to create matrices or flattening matrices. These can make the syntax of comprehensions complex and difficult to read.

In addition, list comprehensions should not be used for huge datasets such as several *million* records as list comprehensions stores the entire output list in memory. Even if your computer has large amounts of RAM, your computer will become non-responsive for several seconds as Python is trying to create the huge list. The alternative is to use a *generator* construct which will learn later.

5.14 References

1. Built-in Functions, https://docs.python.org/3/library/functions.html#sorted
2. Sequence Types — list, tuple, range, https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range
3. Lubanovic, 2019, 'Chapter 7. Tuples and Lists' in Introducing Python, 2nd Edition, O`Reilly Media, Inc.
4. Timmins, 2019, When to Use a List Comprehension in Python, https://realpython.com/list-comprehension-python/#using-list-comprehensions