Chapter 31
Collections, Tuples and Lists

31.1 Introduction

Earlier in this book we looked at some Python built-in types such as `string`, `int` and `float` as well as `bools`. These are not the only built-in types in Python; another group of built-in types are collectively known as collection types. This is because they represent a collection of other types (such as a collection of strings, or integers).

A collection is a single object representing a group of objects (such as a list or dictionary). Collections may also be referred to as *containers* (as they contain other objects). These collection classes are often used as the basis for more complex or application specific data structures and data types.

These collection types support various types of data structures (such as lists and maps) and ways to process elements within those structures. This chapter introduces the Python Collection types.

31.2 Python Collection Types

There are four classes in Python that provide container like behaviour; that is data types for holding collections of other objects, these are

- **Tuples** A Tuple represents a collection of objects that are ordered and immutable (cannot be modified). Tuples allow duplicate members and are indexed.
- **Lists** Lists hold a collection of objects that are ordered and mutable (changeable), they are indexed and allow duplicate members.
- **Sets** Sets are a collection that is unordered and unindexed. They are mutable (changeable) but do not allow duplicate values to be held.
- **Dictionary** A dictionary is an unordered collection that is indexed by a *key* which references a *value*. The value is returned when the *key* is provided. No

J. Hunt, *A Beginners Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-20290-3_31

duplicate keys are allowed. Duplicate values are allowed. Dictionaries are mutable containers.

Remember that everything in Python is actually a type of object, integers are instances/objects of the type `int`, strings are instances of the type `string` etc. Thus, container types such as `Set` can hold collections of any type of thing in Python.

31.3 Tuples

Tuples, along with Lists, are probably one of Pythons most used container types. They will be present in almost any non-trivial Python program.

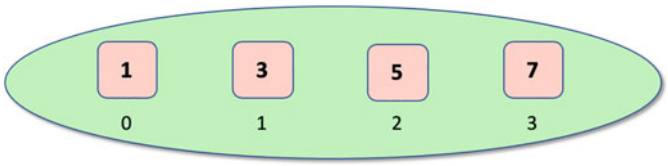
Tuples are an immutable ordered collection of objects; that is each element in a tuple has a specific position (its index) and that position does not change over time. Indeed, it is not possible to add or remove elements from the tuple once it has been created.

31.3.1 Creating Tuples

Tuples are defined using parentheses (i.e. round brackets '`()`') around the elements that make up the tuple, for example:

```
tup1 = (1, 3, 5, 7)
```

This defines a new `Tuple` which is referenced by the variable `tup1`. The `Tuple` contains exactly 4 elements (in this case integers) with the first element in the tuple (the integer 1) having the index 0 and the last element in the `Tuple` (the integer 7) having the index 3. This is illustrated below:



31.3.2 The tuple() Constructor Function

The `tuple()` function can also be used to create a new tuple from an iterable. An iterable is something that implements the iterable protocol (see the last chapter).

This means that a new tuple can be created from a `Set`, a `List`, a `Dictionary` (as these are all *iterable* types) or any type that implements the iterable protocol.

The syntax of the `tuple()` function is:

```
tuple(iterable)
```

For example:

```
list1 = [1, 2, 3]
t1 = tuple(list1)
print(t1)
```

which generates the output:

```
(1, 2, 3)
```

Note that in the above the square brackets are used to represent a *list* of things (the List container type is described in more detail later in this chapter).

31.3.3 Accessing Elements of a Tuple

The elements of a `Tuple` can be accessed using an *index* in square brackets. The index returns the object at that position, for example:

```
print('tup1[0]:\t', tup1[0])
print('tup1[1]:\t', tup1[1])
print('tup1[2]:\t', tup1[2])
print('tup1[3]:\t', tup1[3])
```

which generates the output

```
tup1[0]:      1
tup1[1]:      3
tup1[2]:      5
tup1[3]:      7
```

31.3.4 Creating New Tuples from Existing Tuples

It is also possible to return what is known as a slice from a `Tuple`. This is a new `Tuple` which is comprised of a subset of the original `Tuple`. This is done by

providing the start and end indexes for the slice, separated by a colon, within the index square brackets. For example:

```
print('tup1[1:3]:\t', tup1[1:3])
```

Which returns a new `Tuple` of two elements containing the elements from index 1 up to (but not including) element 3. Note that the original `Tuple` is not affected in any way (remember its immutable so cannot be modified). The output of the above is thus:

```
tup1[1:3]:    (3, 5)
```

There are in fact numerous variations on the use of the slicing indices. For example, if the first index is omitted it indicates that the slice should start from the beginning of the tuple, while omitting the last index indicates it should go to the end of the Tuple.

```
print('tup1[:3]:\t', tup1[:3])
print('tup1[1:]:\t', tup1[1:])
```

which generates:

```
tup1[:3]:     (1, 3, 5)
tup1[1:]:     (3, 5, 7)
```

You can reverse a `Tuple` using the `::` − `1` notation (again this returns a new `Tuple` and has no effect on the original `Tuple`):

```
print('tup1[::-1]:\t', tup1[::-1])
```

This thus produces:

```
tup1[::-1]:   (7, 5, 3, 1)
```

31.3.5 Tuples Can Hold Different Types

Tuples can also contain a mixture of different types; that is they are not restricted to holding elements all of the same type. You can therefore write a `Tuple` such as:

```
tup2 = (1, 'John', Person('Phoebe', 21), True, -23.45)
print(tup2)
```

Which produces the output:

```
(1, 'John', <__main__.Person object at 0x105785080>, True,
-23.45)
```

31.3.6 Iterating Over Tuples

You can iterate over the contents of a `Tuple` (that is process each element in the `Tuple` in turn). This is done using the `for` loop that we have already seen; however, it is the `Tuple` that is used for the value to which the loop variable will be applied:

```
tup3 = ('apple', 'pear', 'orange', 'plum', 'apple')
for x in tup3:
  print(x)
```

This prints out each of the elements in the Tuple `tup3` in turn:

```
apple
pear
orange
plum
apple
```

Note that again the order of the elements in the `Tuple` is persevered.

31.3.7 Tuple Related Functions

You can also find out the length of a `Tuple`

```
print('len(tup3):\t', len(tup3))
```

You can count how many times a specified value appears in a `Tuple` (remember Tuples allow duplicates);

```
print(tup3.count('apple')) # returns 2
```

You can also find out the (first) index of a value in a `Tuple`:

```
print(tup3.index('pear')) # returns 1
```

Note that both `index()` and `count()` are methods defined on the class `Tuple` where as `len()` is a function that the tuple is passed into. This is because `len()` is a generic function and can be used with other types as well such as strings.

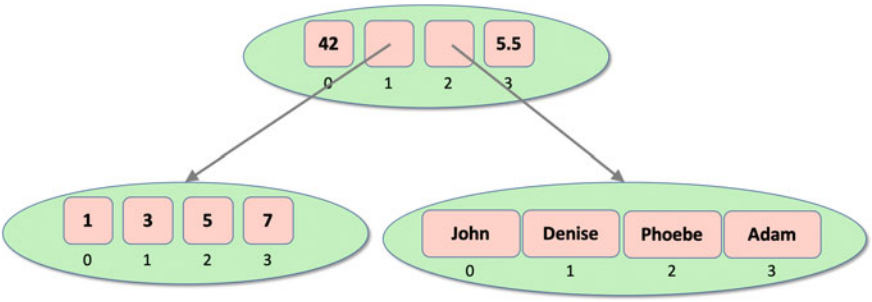
31.3.8 Checking if an Element Exists

You can check to see if a specific element exists in a `Tuple` using the `in` operator, for example:

```
if 'orange' in tup3:
    print('orange is in the Tuple')
```

31.3.9 Nested Tuples

Tuples can be nested within Tuples; that is a `Tuple` can contain, as one of its elements, another `Tuple`. For example, the following diagram illustrates the nesting of a tree of Tuples:



In code we could define this structure as:

```
tuple1 = (1, 3, 5, 7)
tuple2 = ('John', 'Denise', 'Phoebe', 'Adam')
tuple3 = (42, tuple1, tuple2, 5.5)
print(tuple3)
```

The output from this is:

```
(42, (1, 3, 5, 7), ('John', 'Denise', 'Phoebe', 'Adam'), 5.5)
```

Note the *nesting* of round brackets in the printout illustrating where one Tuple is contained within another.

This feature of Tuples (and other containers) allows for arbitrarily complex data structures to be constructed as required by an application.

In fact, a `Tuple` can have nested within it not just other Tuples but any type of container, and thus it can contain Lists, Sets, Dictionaries etc. This provides for a huge level of flexibility when constructing data structures for use in Python programs.

31.3.10 Things You Can't Do with Tuples

It is not possible to add or remove elements from a `Tuple`; they are *immutable*. It should be particularly noted that none of the functions or methods presented above actual change the original tuple they are applied to; even those that return a subset of the original Tuple actually return a new instance of the class `Tuple` and have no effect on the original Tuple.

31.4 Lists

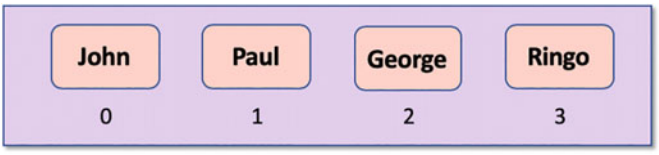
Lists are mutable ordered containers of other objects. They support all the features of the `Tuple` but as they are mutable it is also possible to add elements to a `List`, remove elements and modify elements. The elements in the list maintain their order (until modified).

31.4.1 Creating Lists

Lists are created using square brackets positioned around the elements that make up the `List`. For example:

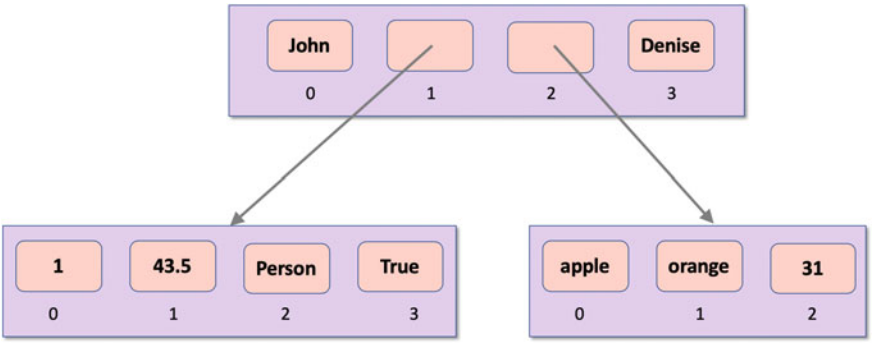
```
list1 = ['John', 'Paul', 'George', 'Ringo']
```

In this case we have created a list of four elements with the first element being indexed from Zero, we thus have:



As with Tuples we can have nested lists and lists containing different types of elements.

We can thus create the following structure of nested Lists:



In code this can be defined as:

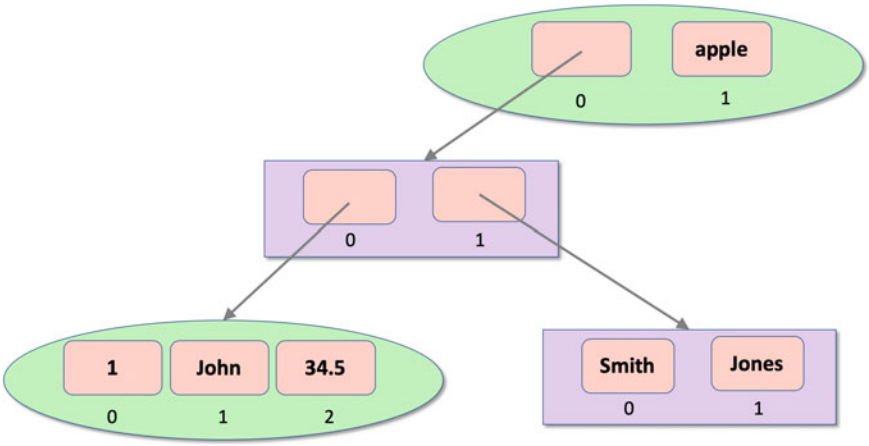
```
l1 = [1, 43.5, Person('Phoebe', 21), True]
l2 = ['apple', 'orange', 31]
root_list = ['John', l1, l2, 'Denise']
print(root_list)
```

When the `root_list` is printed we get

```
['John', [1, 43.5, <tuples.Person object at 0x1042ba4a8>,
True], ['apple', 'orange', 31], 'Denise']
```

Note the square brackets inside the outer square brackets indicating nested lists.

We can of course also nest Tuples in lists and lists in Tuples. For example, the following structure shows Tuples (the ovals) hold references to Lists (the rectangles) and vice versa:



In code this would look like:

```
t1 = (1, 'John', 34.5)
l1 = ['Smith', 'Jones']
l2 = [t1, l1]
t2 = (l2, 'apple')
print(t2)
```

which produces

```
([(1, 'John', 34.5), ['Smith', 'Jones']], 'apple')
```

31.4.2 List Constructor Function

The `list()` function can be used to construct a list from an iterable; this means that it can construct a list from a `Tuple`, a `Dictionary` or a `Set`. It can also construct a list from anything that implements the iterable protocol.

The signature of the `list()` function is

```
list(iterable)
```

For example:

```
vowelTuple = ('a', 'e', 'i', 'o', 'u')
print(list(vowelTuple))
```

produces

```
['a', 'e', 'i', 'o', 'u']
```

31.4.3 Accessing Elements from a List

You can access elements from a list using an index (within square brackets). The index returns the object at that position, for example:

```
list1 = ['John', 'Paul', 'George', 'Ringo']
print(list1[1])
```

This will print out the element at index 1 which is Paul (lists are indexed from Zero so the first element is the zeroth element).

If you use a negative index such as −1 then the index is reversed so an index of −1 starts from the end of the list (−1 returns the last element, −2 the second to last etc.).

It is also possible to extract a slice (or sublist) from a list. This is done by providing a starting and end index to within the square brackets separated by a colon. For example `[1:4]` indicates a slice starting at the *oneth* element and extending up to (but not including) the fourth element. If either of the indexes is missed for a slice then that indicates the start or end of the list respective.

The following illustrates some of these ideas:

```
list1 = ['John', 'Paul', 'George', 'Ringo']
print('list1[1]:', list1[1])
print('list1[-1]:', list1[-1])
print('list1[1:3]:', list1[1:3])
print('list[:3]:', list1[:3])
print('list[1:]:', list1[1:])
```

Which produces:

```
list1[1]: Paul
list1[-1]: Ringo
list1[1:3]: ['Paul', 'George']
list[:3]: ['John', 'Paul', 'George']
list[1:]: ['Paul', 'George', 'Ringo']
```

31.4.4 Adding to a List

You can add an item to a list using the `append()` method of the `List` class (this changes the actual list; it does not create a copy of the list). The syntax of this method is:

```
<alist>.append(<object>)
```

As an example, consider the following list of strings, to which we append a fifth string:

```
list1 = ['John', 'Paul', 'George', 'Ringo']
list1.append('Pete')
print(list1)
```

this will generate the output:

```
['John', 'Paul', 'George', 'Ringo', 'Pete']
```

You can also add all the items in a list to another list. There are several options here, we can use the `extend()` method which will add the items passed to it to the end of the list or we can use the += operator which does the same thing:

```
list1 = ['John', 'Paul', 'George', 'Ringo', 'Pete']
print(list1)
list1.extend(['Albert', 'Bob'])
print(list1)
list1 += ['Ginger', 'Sporty']
print(list1)
```

The output from this code snippet is:

```
['John', 'Paul', 'George', 'Ringo', 'Pete']
['John', 'Paul', 'George', 'Ringo', 'Pete', 'Albert', 'Bob']
['John', 'Paul', 'George', 'Ringo', 'Pete', 'Albert', 'Bob',
'Ginger', 'Sporty']
```

Which approach you prefer to use is up to you.
Note that strictly speaking both `extend()` and += take an *iterable*.

31.4.5 Inserting into a List

You can also insert elements into an existing list. This is done using the `insert()` method of the `List` class. The syntax of this method is:

```
<list>.insert(<index>, <object>)
```

The `insert()` method takes an index indicating where to insert the element and an object to be inserted.

For example, we can insert the string 'Paloma' in between the *Zeroth* and *oneth* item in the following list of names:

```
a_list = ['Adele', 'Madonna', 'Cher']
print(a_list)
a_list.insert(1, 'Paloma')
print(a_list)
```

The result is:

```
['Adele', 'Madonna', 'Cher']
['Adele', 'Paloma', 'Madonna', 'Cher']
```

In other words, we have inserted the string 'Paloma' into the index position 1 pushing 'Madonna' and 'Cher' up one in the index within the `List`.

31.4.6 List Concatenation

It is possible to concatenate two lists together using the concatenation operator `'+'`:

```
list1 = [3, 2, 1]
list2 = [6, 5, 4]
list3 = list1 + list2
print(list3)
```

generates

```
[3, 2, 1, 6, 5, 4]
```

31.4.7 Removing from a List

We can remove an element from a `List` using the `remove()` method. The syntax for this method is:

```
<list>.remove(<object>)
```

This will remove the object from the list; if the object is not in the list then an error will be generated by Python.

```
another_list = ['Gary', 'Mark', 'Robbie', 'Jason', 'Howard']
print(another_list)
another_list.remove('Robbie')
print(another_list)
```

The output from this is:

```
['Gary', 'Mark', 'Robbie', 'Jason', 'Howard']
['Gary', 'Mark', 'Jason', 'Howard']
```

31.4.8 The pop() Method

The syntax of the `pop()` method is:

```
a.pop(index=-1)
```

It removes an element from the `List`; however, it differs from the `remove()` method in two ways:

- It takes an index which is the index of the item to remove from the list rather than the object itself.
- The method returns the item that was removed as its result.

An example of using the `pop()` method is given below:

```
list6 = ['Once', 'Upon', 'a', 'Time']
print(list6)
print(list6.pop(2))
print(list6)
```

Which generates:

```
['Once', 'Upon', 'a', 'Time']
a
['Once', 'Upon', 'Time']
```

An overload of this method is just

```
<list>.pop()
```

Which removes the last item in the list. For example:

```
list6 = ['Once', 'Upon', 'a', 'Time']
print(list6)
print(list6.pop())
print(list6)
```

with the output:

```
['Once', 'Upon', 'a', 'Time']
Time
['Once', 'Upon', 'a']
```

31.4.9 Deleting from a List

It is also possible to use the `del` keyword to delete elements from a list.

The `del` keyword can be used to delete a single element or a slice from a list.

To delete an individual element from a list use `del` and access the element via its index:

```
my_list = ['A', 'B', 'C', 'D', 'E']
print(my_list)
del my_list[2]
print(my_list)
```

which outputs:

```
['A', 'B', 'C', 'D', 'E']
['A', 'B', 'D', 'E']
```

To delete a slice from within a list use the `del` keyword and the slice returned from the list.

```
my_list = ['A', 'B', 'C', 'D', 'E']
print(my_list)
del my_list[1:3]
print(my_list)
```

which deletes the slice from index 1 up to (but not including) index 3:

```
['A', 'B', 'C', 'D', 'E']
['A', 'D', 'E']
```

31.4.10 List Methods

Python has a set of built-in methods that you can use on lists.

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

31.5 Online Resources

See the Python Standard Library documentation for:

- https://docs.python.org/3/tutorial/datastructures.html Python Tutorial on data structures.
- https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range for lists and tuples.
- https://docs.python.org/3/tutorial/datastructures.html#tuples-and-sequences the online tuples tutorial.
- https://docs.python.org/3/tutorial/datastructures.html#lists the online List tutorial.

31.6 Exercises

The aim of this exercise is to work with a collection/container such as a list.

To do this we will return to your `Account` related classes.

You should modify your `Account` class such that it is able to keep a history of transactions.

A *Transaction* is a record of a deposit or withdrawal along with an amount.

Note that the initial amount in an account can be treated as an initial deposit.

The history could be implemented as a *list* containing an ordered sequence to transactions. A Transaction itself could be defined by a class with an action (deposit or withdrawal) and an amount.

Each time a withdrawal or a deposit is made a new transaction record should be added to a transaction history list.

Now provide support for iterating through the transaction history of the account such that each deposit or withdrawal can be reviewed. You can do this by implementing the Iterable protocol—refer to the last chapter if you need to check how to do this. Note that it is the transaction history that we want to be able to iterate through—so you can use the history list as the basis of your iterable.

You should be able to run this following code at the end of your Accounts application:

```
for transaction in acc1:
    print(transaction)
```

Depending upon the exact set of transactions you have performed (deposits and withdrawals) you should get a list of those transactions being printed out:

```
Transaction[deposit: 10.05]
Transaction[deposit: 23.45]
Transaction[withdraw: 12.33]
```