Data Structures in Python

Guillaume Macneil

October 6, 2020

Abstract

This is the fourth lesson on the 'Introduction to Python' course. This course is loosely based around the 'MIT Introduction to Computer Science - Fall 2016' course. The course is more focused towards the Python programming side of things.

1 Introduction to Data Structures

In prior lessons, we have shown how variables can be very useful as they can: **1.** store values **2.** have values re-assigned to them multiple times **3.** act as a further layer of abstraction above just operating with the plain values. We quickly run into an issue though, what if I want to store *multiple values* under one name? To do this, we'll need to use some other Python data structures.

1.1 Tuples

Tuples are an ordered sequence of elements, each of which can have a different type. Each element is *immutable* - the value of each element cannot be edited after assignment.

```
assortment = (623, "Hello", 42.1, True)

print(assortment[0])
# Prints 623
print(assortment[1:3])
# Prints ("Hello", 42.1)

len(assortment)
# Returns 4

assortment = assortment + (False, "Another")
# The "+=" operator could (and should) be used here
print(assortment)
# Prints (623, "Hello", 42.1, True, False, "Another")
```

This may at first seem pretty confusing (that's normal) but the code is actually relatively simple. First, we began with creating a tuple called *"assortment"* and assigned it 4 values. After that, we used a method of accessing the elements of the tuple called *indexing* - we specify the name of the data structure we want to index and then enclosed the index we wanted to access in square brackets, like this *a[3]*. Thirdly, we used another method of accessing elements in the tuple called *slicing*. This is done in a similar way to indexing, only instead of specifying a single index within the square brackets, we specify a range like this *a[2:5]*. After that, we used the *len()* function to return the length of the tuple and concatenated two tuples together, making one large tuple. It is important to note that these operations are not unique tuples, they can be used on many data structures.

1.2 Lists

Much like tuples, lists are ordered sequences of elements accessible by index. The elements in a list are usually homogeneous (of the same type), though they do not have to be. Unlike tuples, the elements of a list are mutable.

```
a = [23, 54, 76, 12, 5, 7, 23, 342]
b = [True, False, "Yes", 0, ["A", ["B", 56.7]]]

len(a)
# Returns 8
len(b)
# Returns 5

print(a[3])
# Prints 12
print(b[3:])
# This '[3:]' notation says 'slice from the fourth element onward'
# Prints [0, ["A", ["B", 56.7]]]

a[0] = "An obvious change"
print(a)
# Prints ["An obvious change", 54, 76, 12, 5, 7, 23, 342]
```

As you can see, lists look quite similar to tuples, with only a few minor differences. Lists are bound by square brackets and, as you can see, can contain any combination of objects. They can be indexed to access individual elements and can be sliced just like tuples. However, there are a few differences. Elements can be reassigned just like a normal variable, by specifying the desired index and using the *'='* operator. Another interesting thing is that you can nest lists (you can actually do the same with tuples too) - you can have a list within a list, or in list *'b'*, a list within a list within a list.

1.3 Dictionaries

Dictionaries are slightly different to the other data structures in the sense that instead of just holding a sequence of elements, dictionaries hold a series of key-value pairs. This means you can query a key and access a value.

```
fruit_stock = {"apples": 43, "bananas": 76, "oranges": 90}

print(fruit_stock["apples"])
# Prints 43
print(fruit_stock["oranges"])
# Prints 90

print("apples" in fruit_stock)
# Prints True
print("cherries" in fruit_stock)
# Prints False

fruit_stock["apples"] = 42
print(fruit_stock["apples"])
# Prints 42
```

Now this does, at first glance, look quite different. The dictionary is bound by curly brackets, instead of there being one object per element, there's two separated by a colon! This is an important feature of dictionaries, the object to the left of the colon is called the *key* and the object to the right is called the *value*. These function in much the same way as actual, physical dictionaries. You search for a word in a certain language, you find it and then get the equivalent word in the other language. In Python dictionaries, you index the dictionary with the key (*a["key"]*) and it returns the value associated with that key. Another thing to note is that dictionaries are also mutable, so you can re-associate a given key with a new value.

2 Data Structure Methods

The fun with data structures doesn't stop there though! Let's suppose that we wanted to add a new element to a list, would we have to completely re-define the list with a new element on the end? That seems like a pretty inefficient, no? Luckily, there is a solution to this - methods.

2.1 Tuple Methods

Tuples only have two methods because they are fundamentally simple data structures that, honestly, can't do very much due to their immutability.

Method:	Parameters:	Example:	Purpose:
a.count()	element	a.count("apple")	Counts the number of instances of a given element in a tuple
a.index()	element (start) (end)	a.index("apple")	Returns the index of a given element in a tuple

2.2 List Methods

Lists, unlike tuples, have many methods (as you can see). In many respects, most of the functionality of lists is due to the methods that can be performed on them.

Method:	Parameters:	Example:	Purpose:
a.append()	item	a.append("apple")	Adds a given item to the end of the list
a.clear()		a.clear()	Empties a given list
a.copy()		a.copy()	Duplicates a given list
a.count()	element	a.count("apple")	Counts the number of instances of an element in a given list
a.extend()	iterable	a.extend(b)	Adds the elements of an iterable to the end of the list
a.index()	element (start) (end)	a.index("apple")	Returns the index of a given element in a list
a.insert()	index element	a.insert(2, "apple")	Inserts an element to the list at a specified index
a.pop()	index	a.pop(3)	Removes and returns an element from a list at a given index
a.remove()	element	a.remove("apple")	Removes the first matching element from a list
a.reverse()		a.reverse()	Reverses the elements of a list
a.sort()	(reverse) (key)	a.sort()	Sorts the elements of a list in ascending or descending order

2.3 Dictionary Methods

Dictionaries, much like lists, also have many methods which allow for the manipulation and collection of data from the dictionary.

Method:	Parameters:	Example:	Purpose:
a.clear()		a.clear()	Removes all items from the dictionary
a.copy()		a.copy()	Makes a shallow copy of the dictionary
a.fromkeys()	sequence (value)	a.fromkeys(k)	Creates a new dictionary from the given sequence of elements
a.get()	key (value)	a.get("apple")	Returns the value for the specified key
a.items()		a.items()	Displays a list of a dictionary's key-value tuple pairs
a.keys()		a.keys()	Displays a list of a dictionary's keys
a.pop()	key (default)	a.pop("apple")	Removes and returns a given element from a dictionary
a.popitem()		a.popitem()	Removes and returns the key-value pair in LIFO order
a.setdefault()	key (default)	a.setdefault("apple")	Returns the value of a key if present, inserts a key-value pair otherwise
a.update()	dictionary	a.update(b)	Updates the dictionary from with elements from another dictionary
a.values()		a.values()	Displays a list of a dictionary's values