Python Chases Monkey

An Introduction to Python Programming

Jason M. Pittman

Chapter 7

Python Makes a Plan

Lists, Arrays, Dictionaries, and Tuples

Introduction

done Can you feel it? The sophistication of our programs is increasing and with it our kung fu repotoire is growing. Likewise, our ability to use what we have learned is getting faster. However, we need to figure out how to manage and organize data so that we can store and retrieve that data without undue complications. Enter, **data structures**.

You see, we need data structures to store and organize data effectively and efficiently. It is true that data structures are complex relative to simple variables. The complexity stems from handling multiple values, mutability, and data relationships. At the same time, data structures are more suitable than simple variables when we need to handle more than a single value at the same time.

Phil the python can feel it. Likewise, Marvin the monkey feels it. The time for the chase draws near. As the trees and their leaves fall silent. The time for organizing and planning is afoot.

Learning Objectives

done 1. Describe commonly used data structure types 2. Articulate performance considerations between various data structure types 3. Develop and implement common data structures using robust and testable code

Data structures

done Python natively recognizes six data structure types along with a variety of subtypes and implementation variations. For example, the **List** data structure is the most common

type. However, a **list** can be implemented as several *queue* types. We are going to focus on three data structure types: *lists*, *tuples*, *dictionaries*. Technically, we could include *sets* in this chapter too but I don't think the differences warrant the extra material.

By way of overview, it is easier to think of all data structures as *collections*. What a specific type of value a given collection can hold is one means of differentiating between data structures. Another way to differentiate between data structures is whether we can change values. That is, some of the data structures are mutable, meaning elements cannot be altered after initialization. Others (read: most) are mutable, thus values in the collection can be added, altered, or removed any time. We'll deal with which is which as we explore each data structure type.

Arrays

done If a **variable** is container for a single value as in `name = "Phil"` then an array is a special kind of variable or collection we can use to hold more than one value.

While not a built-in data structure type, we can use the traditional **Array** by using `import array` module or through an external provider such as *Numpy*.

Confusingly, Python does not- I repeat does not!- have a built-in **Array** data structure type without help from a module. The confusion stems from just about every other programming language including the **Array** type as a basic data structure. Instead, the fundamental data structure type in Python is the **List**. That said, understanding the concept of an **Array** is crucial to coming to realize just how powerful the **List** type is in comparison as well as understanding something deeper about one of the primitive data types we dicussed in an earlier chapter.

Now, with that said, the real reason for mentioning **arrays** as a *collection* concept is so we can experience two things. First, arrays and thus collections are zero-indexed. Thus, the index of the first element in the array or collection is `0`. Second, accessing elements (the *values* in these arrays and collections) using the index has a particular syntax using brackets like `[index]`.

Let's check out a specific and simple case.

Strings

done A quick aside and word on *strings*. Remember when we defined a **str** type as belonging the category of data types called *Text*? Well, the `str` is also a type of generic **Array**. If we consider that Python is built on a `C` foundation, and we know that `C` can do an array of characters (e.g., letters) but not a **string**, this will make way more sense.

Consider the following when we have the assignment statement `name = "Phil"`: #### Example 1

```
(1) print(name[0])
```

You guessed it, line one will output `P`. The reason we get the single character `P` is due to the string also functioning as a character array. Furthermore, we can trust that we will always get `P` as long as a new value isn't assigned to the variable `name` because the **string** is immutable. We can also trust that we will always get a character- more appropriately, a *substring* of the string- because arrays can only hold values of the same type.

Pretty cool, right? All right, we can do even cooler things using the next data structure type.

Lists

done What most people think of when they are implementing a collection (or more specifically, an **Array**) in Python is actually the **List** data structure type. In brief, a **List** is a data structure capable of holding *arbitrary* elements. The arbitrariness applies to both the length of the collection as well as the types of values held in the collection.

Example 2

```
(1) python = [1, False, "Peter", 2.75]
```

One of the very unique things about lists in Python is that we can use them to hold different types. The reason we can do this is because everything is just an object in Python, remember. I mentioned this earlier in the book and told you we'd come back to the point. The significance here is that we can fully pack `python` collection with multi-type values representing our snake friend. In fact, we can *pack* a list with snake objects:

Example 3

```
(1) phil = Snake("Phil")
(2) piotr = Snake("Piotr")
(3) peetie = Snake("Peetie")
(4)
(5) snakes = [phil, piotr, peetie]
```

By packing our snake objects into a collection like this, now we can deal with a single object instead of three separate objects if we so choose. To add to the significance, because a **List** is mutable, we can modify the collection during runtime. Let's say we have a fourth snake, `paula = Snake("Paula")`, and we want to add this new danger nooble to the collection. Well:

Example 4

```
(1) snakes.append(paula)
```

The `paula` snake will be added to the end of the collection. If for some reason we want `paula` to be added at a specific index, we can pass `paula` as a parameter to the `.insert(index, value)` method while `index` is the index as an integer where we want `value` inserted.

Another cool thing we should look at briefly is how to access the last element in a **List** regardless of its length. If `[0]` gets us the first element, we simply need to *wrap around* to the other end using `[-1]`.

Building on this, we can return the entire **List** in reverse order if we call `snakes[::-1]`. This actually reveals two new aspects of the **List** type. Order of elements is an underlying principle which is why we can reverse output them. As well, the `:` syntax introduces the concept of *slicing*. For example, if we call `snakes[0:1]1` we will return just the initial two elements. Neat.

By the way, as we might have experienced in some labs or projects, the **List** data structure pairs well with **definite** iteration. Using the `in` keyword for instance becomes quite powerful when we nest a list within a list. Sound strange? Well imagine we `mike` the `Monkey()` wanted to fend off our four snakes. The monkey needs to know how much *health* each snakes has, right?

Example 5

```
(1) snakes = [[phil, 10], [piotr, 20], [peetie, 5]]
```

If we call `snakes[1][1]`, we will get the integer value `20` associated with the snake `piotr`. Understand, there are no language limitations to how long the **List** or nested **List** can be although we do need to be mindful of program efficiency constraints. Speaking of constraints, what if we had a constraint whereby we didn't want a *list** to be modified after declaration?

To say that the **List** type has a plethora of methods available to access or otherwise modify the contents of the collection would be an understatement in any case. We have explored a standard set to get started; I encourage future exploration of the full set of methods available as our programs grow in complexity.

Tuples

We can describe a **Tuple** as a the combination of a **string** as a collection and a **List**. In other words, a **Tuple** is an immutable **List**. Visually, we can differentiate between a tuple and a list by looking at the value bounding characters. Whereas a list uses square brackets, a tuple is defined by values encased in parentheses. Apart from that, everything available with a **List** collection is also available to a **Tuple**.

Example 6

```
(1) phil = Snake("Phil")
(2) piotr = Snake("Piotr")
(3) peetie = Snake("Peetie")
(4)
(5) snakes = (phil, piotr, peetie)
```

See? The concept is the same as in **Example 4** which was a **List** except for the parentheses! Just remember, we cannot modify a tuple after it is constructed. Along those lines, there might be a temptation to run something like `tuple + (value,)`, don't give in. This works but only because a copy of the original is constructed which increases the space complexity of our program.

Dictionaries

The **Dictionary** is the real power move in our data structure kung fu. To demonstrate both this power and the differences between the **List** and **Dictionary** types, let's look at an example conversion of the `snakes` collection.

Example 7

```
(1) phil = Snake("Phil")
(2) piotr = Snake("Piotr")
(3) peetie = Snake("Peetie")
(4)
(5) snakes = {
(6)    1: phil,
(7)    2: piotr,
(8)    3: peetie
(9) }
```

First, we see the **Dictionary** uses curly braces instead of parenthese or square brackets. That's a good visual clue. Second, we see that our snakes have some kind of index paired to them such as `'1': phil`. This is a *key* and *value* pairing and is the unique feature of the dictionary.

In **Example 7** it just so happens we use an **int** for our keys but the data type for our keys and values is not restricted. Think about the importance of this for a moment.

While we think, I'd like to point out there are a handful of ways to define our dictionaries. **Example 7** is the most explicit given lines five through nine. Alternatively, and perhaps somewhat more Pythonically, we could articulate the same idea one a single line.

Example 8

```
(1) snakes = {1: phil, 2: piotr, 3: peetie}
```

As long as we pay attention to the dictionary syntax, I suggest **Example 8** is more readable than **Example 7**. Yet, there are no efficiency differences. Anyway, now is a good time to discuss how we can access our keys and values.

If we simply call `snakes`, we will get the entire dictionary as output as key-value pairs. Sometimes we want the entire dictionary but often we will want specific data (either to return for output or to modify). Consider this example:

Example 9

```
(1) print(snakes[1])
```

Access a specific key will return the corresponding mapped value. If we try, say `print(snakes[phil])` Python will raise a `KeyError` exception. The basis for the exception is that dictionary values are only accessible through keys. Why? Well, because values are not unique in a dictionary, unlike keys.

Likewise, if we want to add a key-value pair to an existing dictionary like `snakes`, we can do something like `snakes[4] = pattie` as long as we know the key is unique. Fortunately, the syntax to change an existing value is identical but something we ought to watch out for so we do not inadvertantly overwrite data.

Okay, done thinking about the importance of key and value data types? I'll tell you want I think: it seems to me that we can leverage the power of *mapping* some key to some value such that the value is also a data structure. Check this out:

Example 10

```
(1) plan = {
(2)     monkeys: [mike, mary, martie],
(3)     plants: [tree, bush],
(4)     wind: True
(5) }
```

Now we have a plan! To build a similar data structure using a **List**, we would need to use nesting. Fair. However, to access or modify such a nested list would come at an efficiency cost. On the contrary, by using a **Dictionary**, we can more efficiently interact with our data. The time and space tradeoff between these data structures can be significant depending on the nature of the data.

One last tidbit regarding the **Dictionary**. Technically, dictionaries are unordered. However, a dictionary is driven by a unique *hash* for each key-value pair in the data structure. This is why (a) a key must be a single immutable value and no two keys in the dictionary can be the same. Thus, if a dictionary exists, we can trust that the key-value pairs are unique.

Exercises

1. Write a simple program which uses **definite** iteration to loop over **Example 1** to reconstruct the original value.
2. Rewrite the program from **Exercise 1** using an **indefinite** loop.
3. Write a unit test to measure the time efficiency of an equivalent **Dictionary** and **List** containing our three monkeys by name.

Questions

1. In **Example 1**, what would `print(name[4])` output?
2. Given our **List** collection of snakes in **Example 3**, which snake would be returned if we call `snakes[-2]`?
3. Can we *convert* between a **tuple** and **list** collection without building a copy of the tuple?