Python Data Structures (Python for Data Science Basics #2)

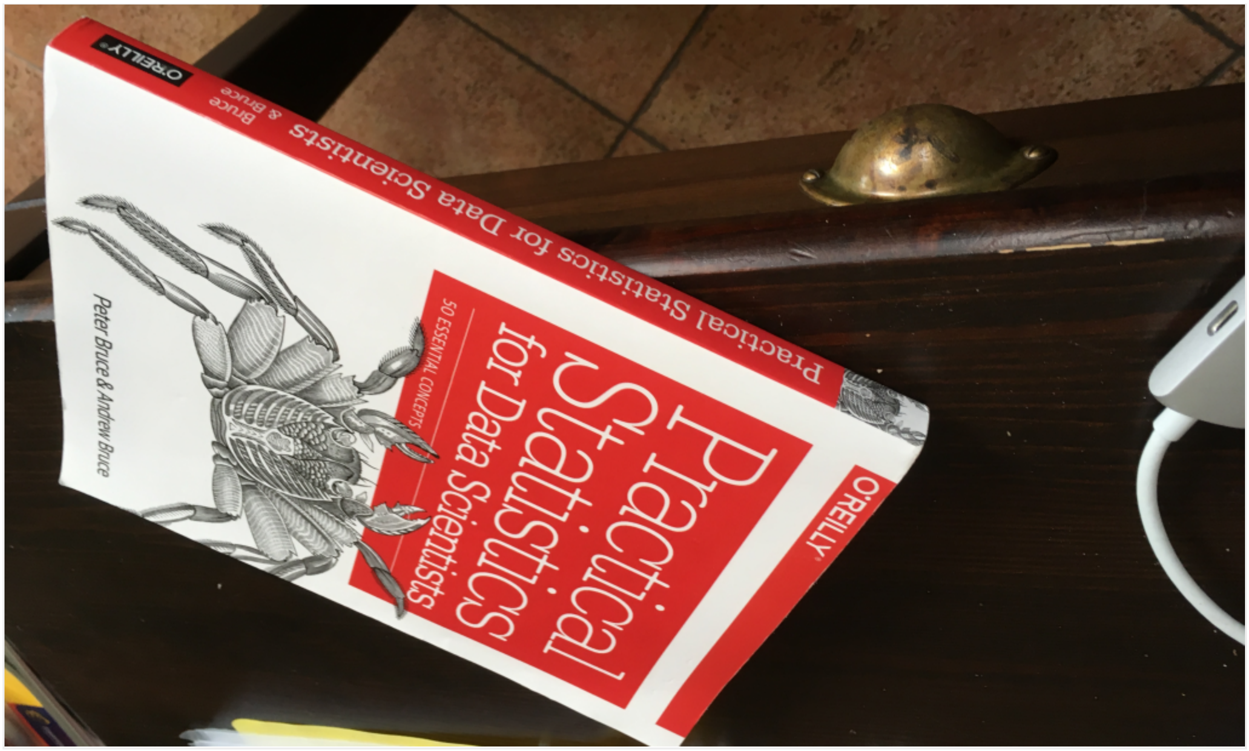
Where did we leave off? Oh, right, we learned about how to use variables in Python. Here is the second essential topic that you have to learn if you are going to use Python as a Data Scientist: **Python Data Structures**!

Note: This is a hands-on tutorial. I highly recommend doing the coding part with me – and if you have time, solving the exercises at the end of the tutorial! If you haven't done so yet, please go through these two first:

1. *How to install Python, R, SQL and bash to practice data science!*
2. *Python for Data Science #1 – Tutorial for Beginners – Python Basics*

Why care about Python Data Structures?

Imagine that you have a book on your desk. I have one on mine: *P. & A. Bruce: Practical Statistics for Data Scientists*. If I want to store this info in Python, I can put it into a variable.


Practical Statistics for Data Scientists
50 ESSENTIAL CONCEPTS
Peter Bruce & Andrew Bruce
O'REILLY

```
my_book = "Practical Statistics for Data Scientists"
```

Done!

But hey, I just missed two more books on the other side of my desk! *Dan Brown: Digital Fortress* and *George R. R. Martin: A Game of Thrones*. How do I store these two new pieces of information… Maybe I can set up two new variables:

```
my_book2 = "Digital Fortress"
```

```
my_book3 = "A Game of Thrones"
```

Wait a minute! I've just realized I have a whole bookshelf behind me…

Are you seeing the problem? Sometimes in Python we need to store relevant information together in one object – instead of several small variables.

This is why we have Data Structures!

Python Data Structures

There are three major **Python data structures**:

- Lists.

  ```
  book_list = ['A Game of Thrones', 'Digital Fortress', 'Practical Statistics for Data Scientists']
  ```
- Tuples.

  ```
  book_tuple = ('A Game of Thrones', 'Digital Fortress', 'Practical Statistics for Data Scientists')
  ```
- Dictionaries

  ```
  book_dictionary = {'George R. R. Martin': 'A Game of Thrones', 'Dan Brown': 'Digital Fortress', 'A. & P. Bruce': 'Practical Statistics for Data Scientists'}
  ```

All three are good for different things and you have to use them slightly differently…

Let’s dig into the practical details!

Python Data Structures #1: List

Start with the simplest one: Python **lists**.

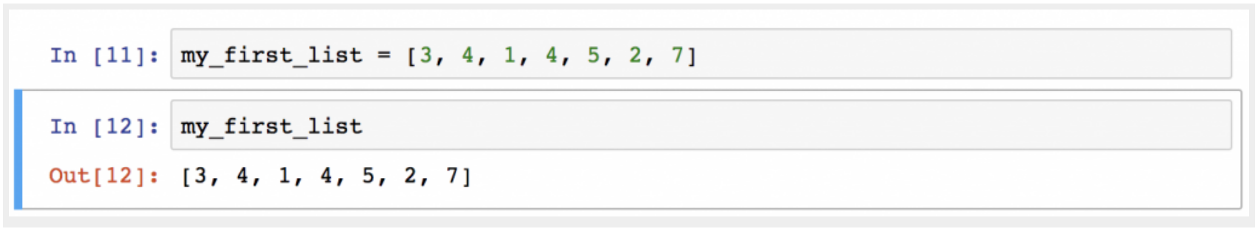
A **list** is a sequence of values. Basically, it's data put into brackets and separated by commas. An easy example – a list of integers:

```
[3, 4, 1, 4, 5, 2, 7]
```

It's important to know that in Python, a list is an object – and generally speaking it's treated like any other data type (e.g. integers, strings, booleans, etc.). This means that you can assign your list to a variable, so you can store and make it easier to access:

```
my_first_list = [3, 4, 1, 4, 5, 2, 7]
```

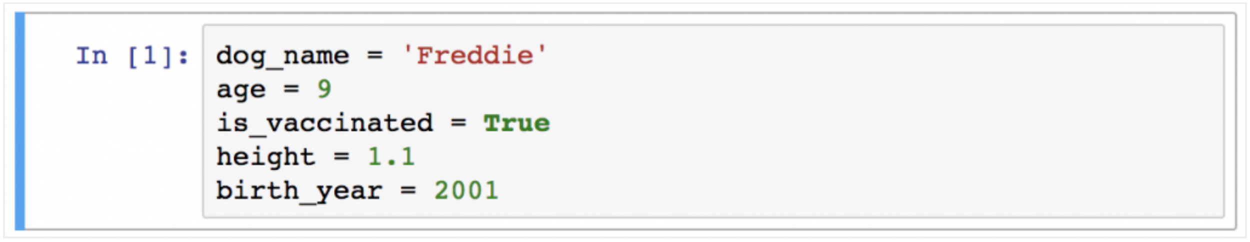
```
my_first_list
```



```
In [11]: my_first_list = [3, 4, 1, 4, 5, 2, 7]

In [12]: my_first_list
Out[12]: [3, 4, 1, 4, 5, 2, 7]
```

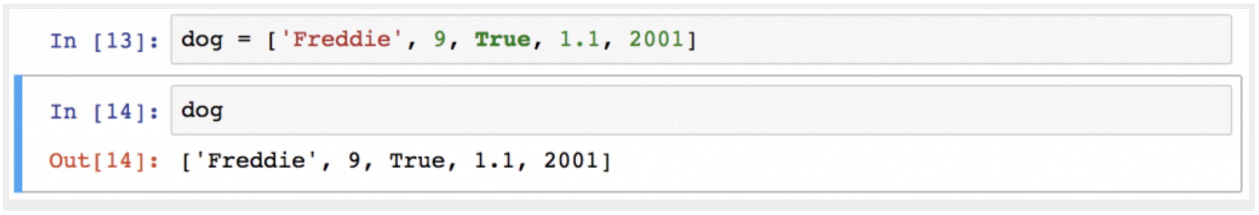
A list can hold every other type of data, not just integers – strings, Booleans, even other lists. Interesting, huh? Do you remember Freddie, the dog from the previous tutorial?



```
In [1]: dog_name = 'Freddie'
        age = 9
        is_vaccinated = True
        height = 1.1
        birth_year = 2001
```

You can store his attributes in one list instead of 5 different variables:

```
dog = ['Freddie', 9, True, 1.1, 2001]
```



```
In [13]: dog = ['Freddie', 9, True, 1.1, 2001]

In [14]: dog

Out[14]: ['Freddie', 9, True, 1.1, 2001]
```

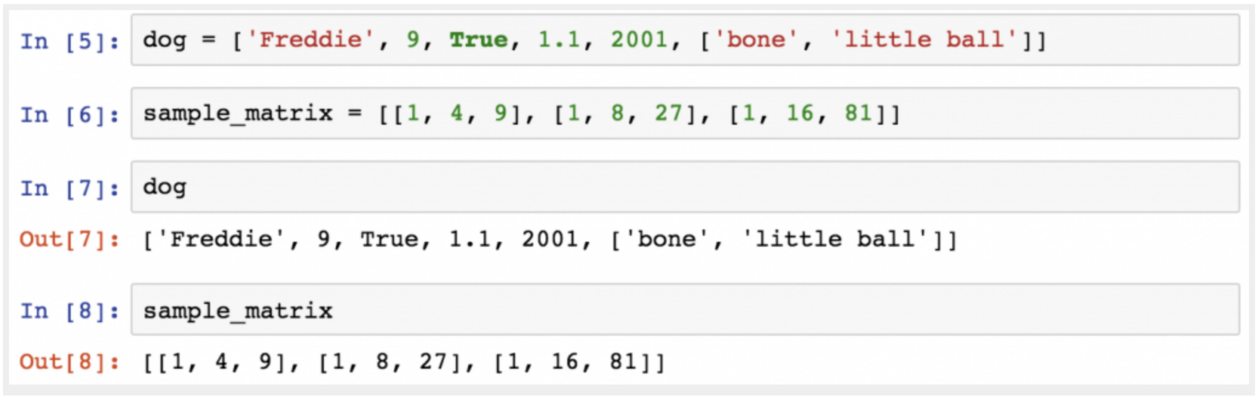
Now let’s say that Freddie has two belongings: a bone and a little ball. We can store those belongings as a list inside our first list.

```
dog = ['Freddie', 9, True, 1.1, 2001, ['bone', 'little ball']]
```

Actually we can do this list-in-a-list thingy infinite times – and believe it or not, this simple concept (the official name is “nested lists,” by the way) will be essential when it comes to the actual Data Science part of Python – e.g. when we create some *multidimensional numpy arrays* to run correlation analyses… but let's not get into it yet! The only thing you should remember is that you can store lists in lists.

Or try this:

```
sample_matrix = [[1, 4, 9], [1, 8, 27], [1, 16, 81]]
```



```
In [5]: dog = ['Freddie', 9, True, 1.1, 2001, ['bone', 'little ball']]

In [6]: sample_matrix = [[1, 4, 9], [1, 8, 27], [1, 16, 81]]

In [7]: dog
Out[7]: ['Freddie', 9, True, 1.1, 2001, ['bone', 'little ball']]

In [8]: sample_matrix
Out[8]: [[1, 4, 9], [1, 8, 27], [1, 16, 81]]
```

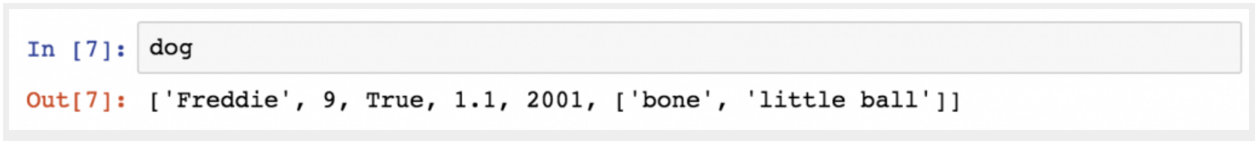
Do you feel scientific? You should, because you have just created a 3-by-3 2D matrix.

How to access a specific element of a Python list?

Now that we have stored these values, it's really essential to know how to access them in the future. As you have already seen, you can get the whole Python list returned if you type the right variable name.

E.g.

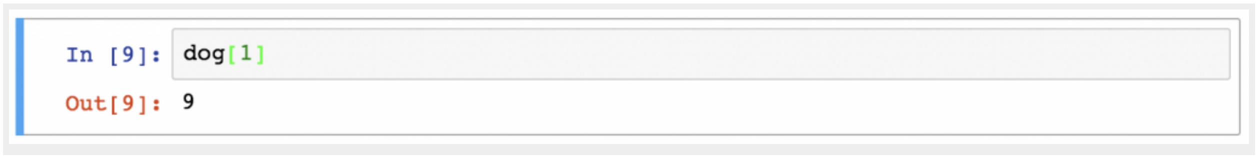
`dog`



```
In [7]: dog
Out[7]: ['Freddie', 9, True, 1.1, 2001, ['bone', 'little ball']]
```

But how do you call one particular item from your list? Firstly, think a bit about how you can refer to a value in theory… The only thing that comes into play is the position of the value. E.g. if you want to call the first element on the `dog` list, you have to type the name of the list and the number of the element between brackets, like this: `[1]`. Try this:

`dog[1]`



```
In [9]: dog[1]
Out[9]: 9
```

What??? `9` was the second element on the list, not the first. Well, not in Python… Python uses so-called "*zero-based indexing*", which means that the first element's number is `[0]`, the second is `[1]`, the third is `[2]` and so on.

This is something you have to keep in mind, when working with Python Data Structures.

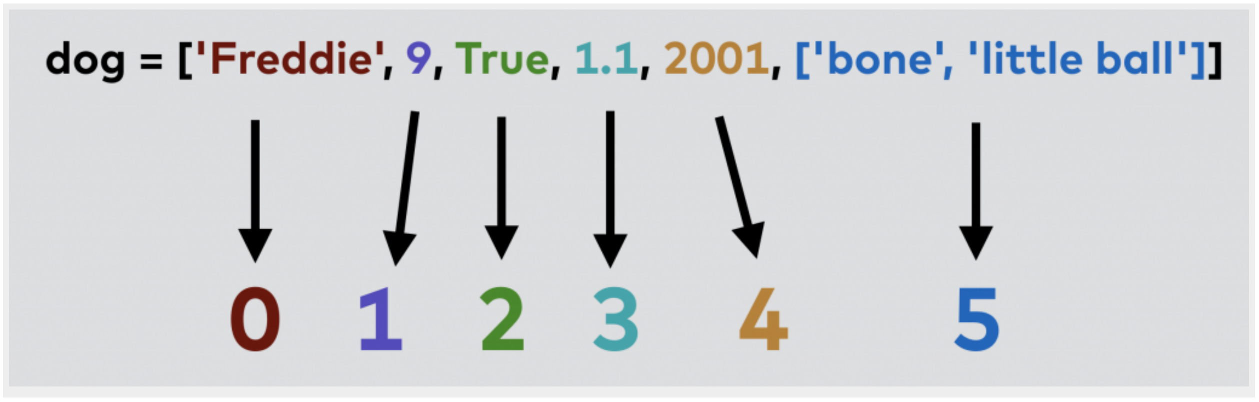
Note: "But why is that?" Hmm, tough topic! I don't dare to say, "because of nerds…" So instead, I'll just link this nice open letter by Prof Dijkstra from 1982: http://www.cs.utexas.edu/users/EWD/transcriptions/EWD08xx/EWD831.html

Anyway, don't think too much about this… Just accept and apply this strange rule!

But here's a detailed example!

Freddie the dog:

```
dog = ['Freddie', 9, True, 1.1, 2001, ['bone', 'little ball']]
```



```
dog = ['Freddie', 9, True, 1.1, 2001, ['bone', 'little ball']]
```

Try to print all the list elements one by one:

`dog[0]`

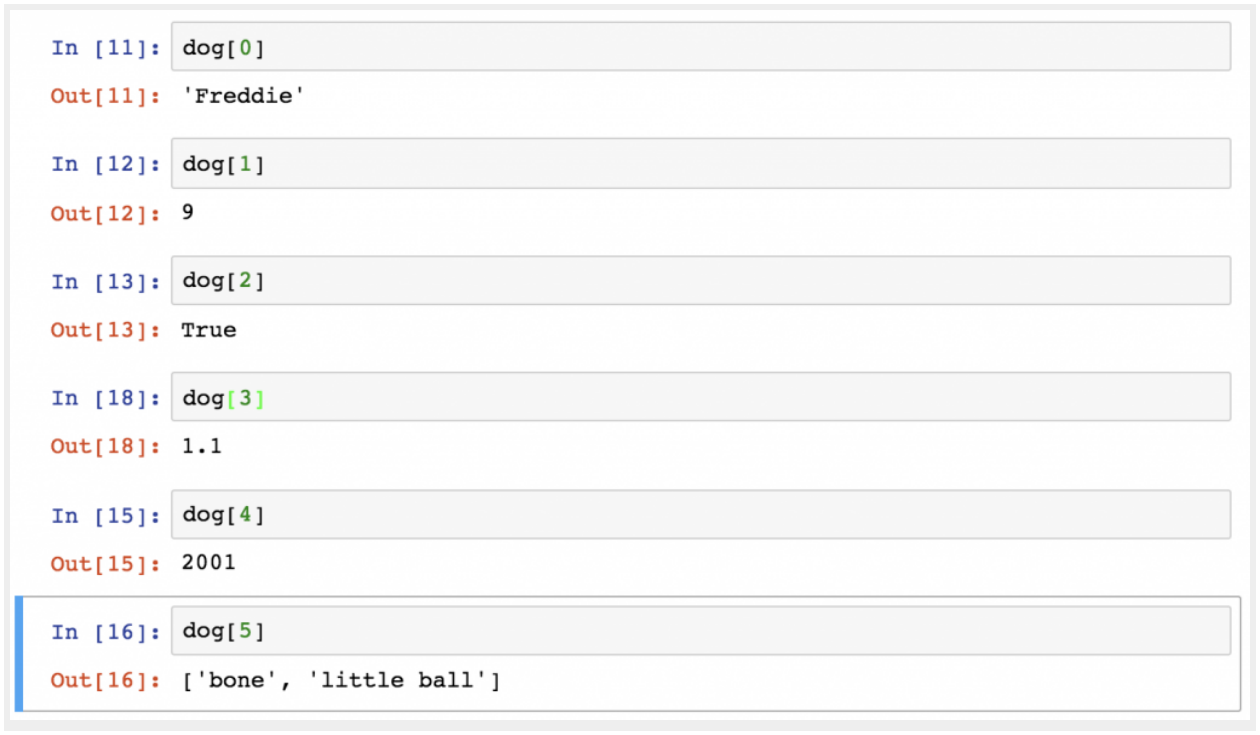
`dog[1]`

`dog[2]`

`dog[3]`

`dog[4]`

`dog[5]`



```
In [11]: dog[0]
Out[11]: 'Freddie'

In [12]: dog[1]
Out[12]: 9

In [13]: dog[2]
Out[13]: True

In [18]: dog[3]
Out[18]: 1.1

In [15]: dog[4]
Out[15]: 2001

In [16]: dog[5]
Out[16]: ['bone', 'little ball']
```

Twisted… But you will get used to it!

How to access a specific element of a nested Python list

One more thing about Freddie! We want to print his belongings one by one!

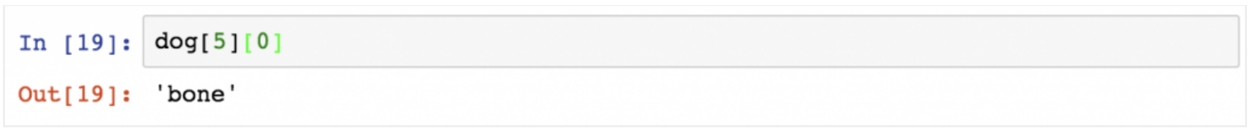
`dog = ['Freddie', 9, True, 1.1, 2001, ['bone', 'little ball']]`

Can you find out how to get the ‘bone’ element, which is located in a nested list? Actually it’s super-intuitive.

It’s gonna be the zeroth element of our fifth element! The syntax is:

`dog[5][0]`

That’s it:



```
In [19]: dog[5][0]
Out[19]: 'bone'
```

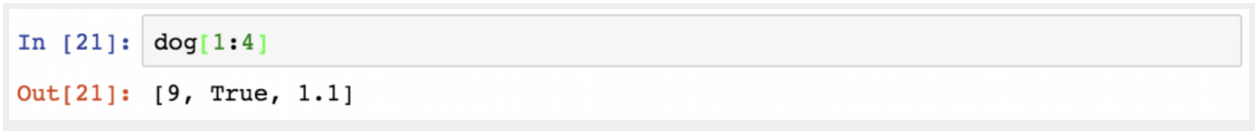
If this is not 100% clear yet, I suggest playing around a bit with the `sample_matrix = [[1, 4, 9], [1, 8, 27], [1, 16, 81]]` data set and you will learn the trick!

How to access multiple elements of a Python list

One more trick you might wanna know about! You can use a colon between two numbers in your brackets, so you can get a sequence of list items.

```
dog = ['Freddie', 9, True, 1.1, 2001, ['bone', 'little ball']]
```

```
dog[1:4]
```



```
In [21]: dog[1:4]
Out[21]: [9, True, 1.1]
```

We'll get back to this feature later in detail!

…

This is everything you need to know about Python lists for now!

Python Data Structures #2: Tuples

What is a Python tuple? First of all: as a junior/aspiring Data Scientist, you don't have to care too much about tuples. If you want you can even skip this section.

If you have stayed:

A Python tuple is almost the same as a Python list, with a few small differences.

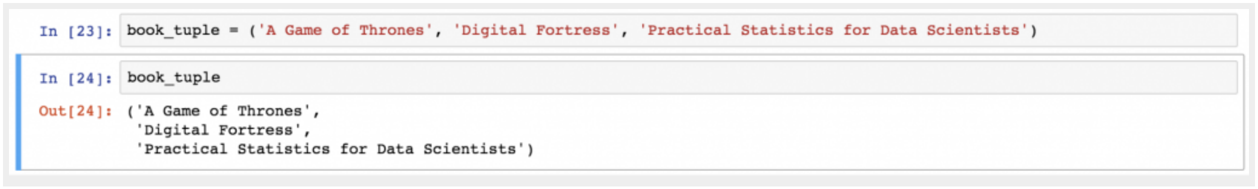
1. Syntax-wise: when you set up a tuple, you won't use brackets, but parentheses.

   List:

   `book_list = ['A Game of Thrones', 'Digital Fortress', 'Practical Statistics for Data Scientists']`

   Tuple:

   `book_tuple = ('A Game of Thrones', 'Digital Fortress', 'Practical Statistics for Data Scientists')`

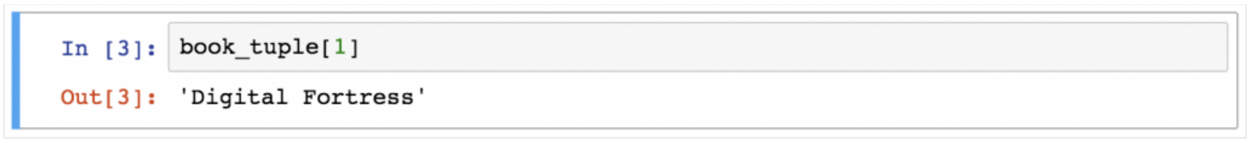
   ```
   In [23]: book_tuple = ('A Game of Thrones', 'Digital Fortress', 'Practical Statistics for Data Scientists')

   In [24]: book_tuple
   Out[24]: ('A Game of Thrones',
             'Digital Fortress',
             'Practical Statistics for Data Scientists')
   ```

2. A Python list is *mutable* – so you can add, remove and change items in it. On the other hand, a Python tuple is *immutable*, so once it's set up, it's sort of "set in stone." This strictness can be handy in some cases to make your code safer.

3. Python tuples are slightly faster than Python lists with the same calculations.

Other than that, you can use a tuple pretty much the same way as a list. Even returning an item happens via the same bracket frames method. (Try: `book_tuple[1]` for your freshly created tuple.)



```
In [3]: book_tuple[1]
Out[3]: 'Digital Fortress'
```

Again: none of the above will be your concern when you just start off with Python coding, but it’s good to know at least this bit about tuples.

Python Data Structures #3: Dictionaries

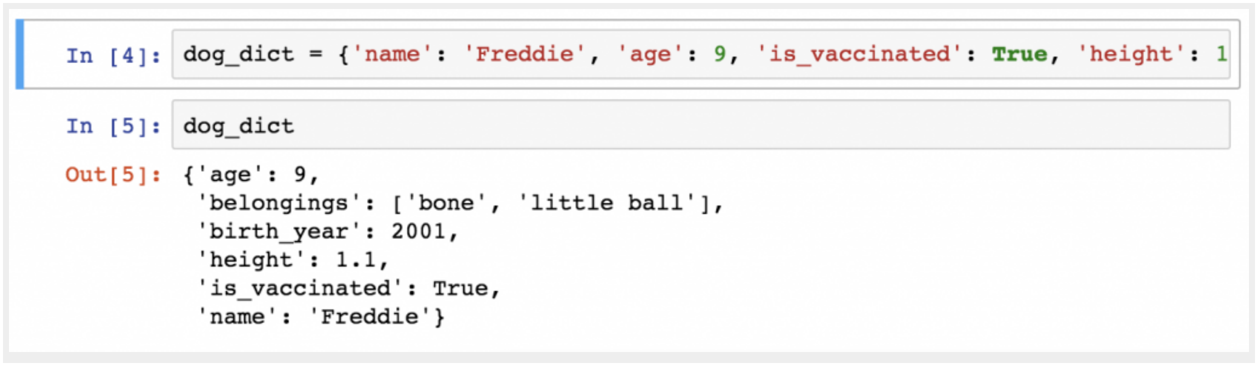
Dictionaries are a whole different story. They are actually very different from lists – and very commonly applied and useful in data science projects.

The main concept of dictionaries is that for every value you have a unique key. Take a look at Freddie the dog again:

`dog = ['Freddie', 9, True, 1.1, 2001, ['bone', 'little ball']]`

These are the **values** that we want to store about a dog. In a dictionary you can attribute a key for each of these values, so you can understand better what value stands for what.

```
dog_dict = {'name': 'Freddie', 'age': 9, 'is_vaccinated': True,
'height': 1.1, 'birth_year': 2001, 'belongings': ['bone', 'little
ball']}
```



```
In [4]: dog_dict = {'name': 'Freddie', 'age': 9, 'is_vaccinated': True, 'height': 1

In [5]: dog_dict

Out[5]: {'age': 9,
         'belongings': ['bone', 'little ball'],
         'birth_year': 2001,
         'height': 1.1,
         'is_vaccinated': True,
         'name': 'Freddie'}
```

As you can see the output is already formatted by Python… For better understanding you can do the same for yourself on the first hand – let’s put the key-value pairs into new lines:

```
dog_dict = {'name': 'Freddie',

'age': 9,

'is_vaccinated': True,

'height': 1.1,
```

`'birth_year': 2001,`

`'belongings': ['bone', 'little ball']}`

As you can see, a nested list (`belongings` in this example) as a value in a dictionary is not a problem.

And this is how a Python dictionary looks!

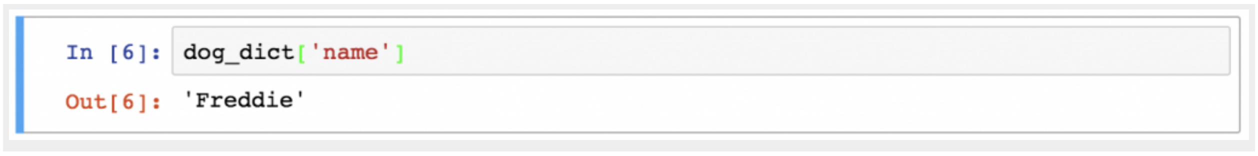
How to access a specific element of a Python dictionary

Here’s the most important rule to remember when it comes to accessing any element of any kind of Python data structure: **whether it’s a list, a tuple or a dictionary, you can print a specific item by typing the name of your data structure (eg.** `dog`**) and the unique identifier of the element between brackets (eg.** `[1]`**).**

The same goes for dictionaries.

The only difference is that while in lists and tuples the unique identifier was the number of the element – in a dictionary it's the key. Try this:

```
dog_dict['name']
```



```
In [6]: dog_dict['name']
Out[6]: 'Freddie'
```

Note 1: maybe you are wondering whether you can still use a number to call a dictionary value. It's not possible, because Python dictionaries are unordered by definition – this means none of the key-value pairs have a number in the dictionary. You can check this right away if you put in a new dictionary – when you call it back to your screen, it'll change your original order to alphabetical order. (Check above!)

Note 2: maybe you are also wondering if you can return a key by inputting a value and not just a value by inputting a key. Bad news: it's not possible – Python is simply not made for that.

Test yourself!

Tadaaa! End of the article! It's time to test yourself! You have learned a lot of important new things about Python Data Structures today. If you haven't been doing the “Test yourself” sections in my articles, please make an exception for this one. Python Data Structures are something that you will use all the time when you work as a Data Scientist, so do yourself a favor and practice a bit to understand the topic 100%!

Here’s the exercise:

Copy-paste this super-nested (and super nasty) Python list-dictionary mutant into your Jupyter notebook and put it into a variable called test!

```
test = [{'Arizona': 'Phoenix', 'California': 'Sacramento', 'Hawaii':
'Honolulu'},
1000,
2000,
3000,
['hat', 't-shirt', 'jeans', {'socks1': 'red', 'socks2': 'blue'}]]
```

1.
2. **Solve these 6 small assignments – by printing specific items from the list/dictionary above! It’ll start easy, then it’s gonna be harder and harder!**

   Exercise #1: Return `2000` on your screen!

   Exercise #2: Return the dictionary of the cities and states on your screen! (This: `{'Arizona': 'Phoenix', 'California': 'Sacramento', 'Hawaii': 'Honolulu'}`)

   Exercise #3: Return the list of the clothes on your screen! (This: `['hat', 't-shirt', 'jeans', {'socks1': 'red', 'socks2': 'blue'}]`)

   Exercise #4: Return the word `'Phoenix'` on your screen!

Exercise #5: return the word `'jeans'` on your screen!

Exercise #6: Return the word `'blue'` on your screen!

.

.

.

Solutions

EXERCISE #1

`test[2]` –» The only trick here is the zero-based indexing, so in our list `2000` is in the 2nd place in Python terms.

EXERCISE #2

`test[0]` –» This will print the whole dictionary from our main list.

EXERCISE #3

`test[4]` –» Same as the previous two – it will print the nested list.

EXERCISE #4

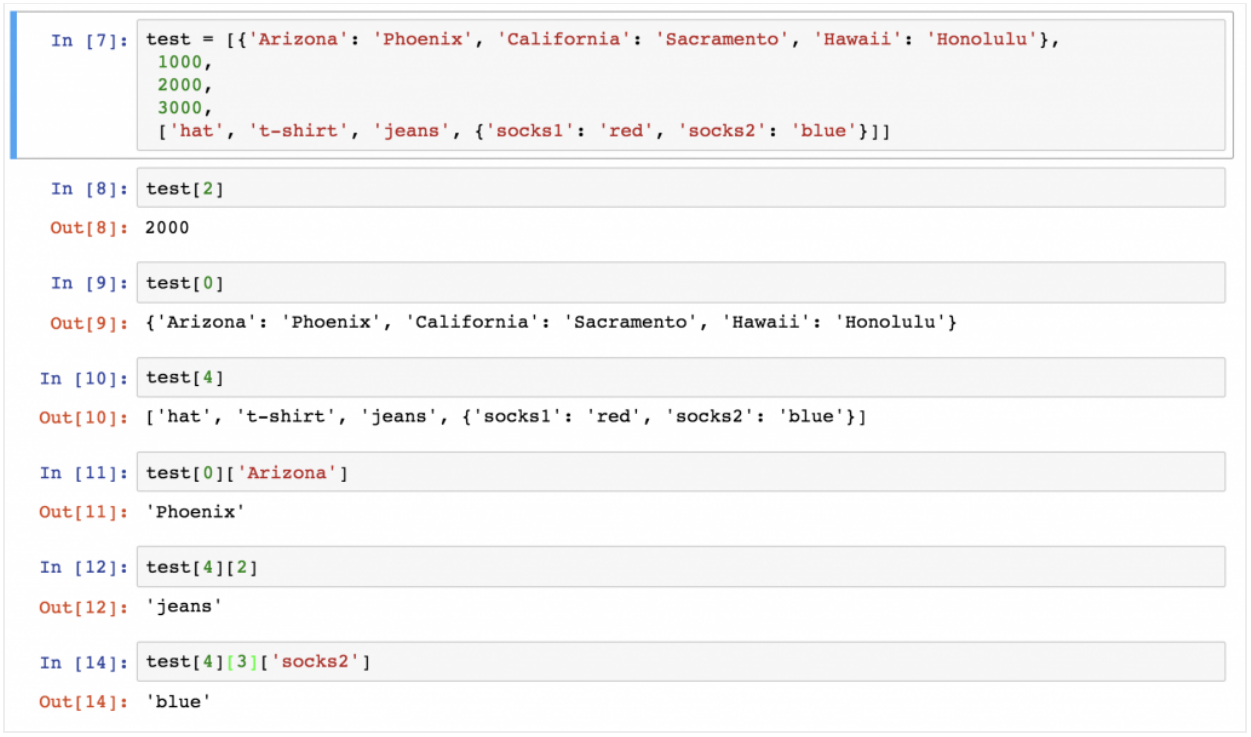
`test[0]['Arizona']` –» This is basically the next step of exercise #2 – we are calling the `'Phoenix'` value with its key: `'Arizona'`.

EXERCISE #5

`test[4][2]` –» And this one is related to exercise #3 – referring to`'jeans'` by its number – don't forget the zero-based indexing.

EXERCISE #6

`test[4][3]['socks2']` –» And one more step – calling the item of a dictionary of a nested list within a list – by its key: `'socks2'`.



```
In [7]: test = [{'Arizona': 'Phoenix', 'California': 'Sacramento', 'Hawaii': 'Honolulu'},
         1000,
         2000,
         3000,
         ['hat', 't-shirt', 'jeans', {'socks1': 'red', 'socks2': 'blue'}]]

In [8]: test[2]

Out[8]: 2000

In [9]: test[0]

Out[9]: {'Arizona': 'Phoenix', 'California': 'Sacramento', 'Hawaii': 'Honolulu'}

In [10]: test[4]

Out[10]: ['hat', 't-shirt', 'jeans', {'socks1': 'red', 'socks2': 'blue'}]

In [11]: test[0]['Arizona']

Out[11]: 'Phoenix'

In [12]: test[4][2]

Out[12]: 'jeans'

In [14]: test[4][3]['socks2']

Out[14]: 'blue'
```

Conclusion

Nice. Job. You are done with another Python tutorial! This is almost everything you have to know about Python Data Structures! Well, in fact, there will be a lot of small, but important details (e.g. how to add, remove and change elements in a list or in a dictionary)… but to get there we need to talk a little bit about Python functions and methods, and some other exciting things first