

# tutorial008-AdvancedDataTypesContainers

September 30, 2022

## 1 Introduction to Advanced Python Data Types

### 1.0.1 Learning Goals

- Advanced python data types
- The concept of data container
- Python Containers
  - Lists
  - Strings
  - Tuples
  - Dictionaries
  - Sets

In the previous tutorial we have focussed the simple data types in python, `int`, `float`, `complex`, `bool`, and `str`. In geenral these simple variables types store individual variable values (`a = 10`; or `b = 2 + 2j`).

If you look back at the previous tutorial, `str` (strings) are a little bit different. They can store multiple characters all at once. For example:

```
[1]: mystring = 'this list contains five words.'  
print(mystring)
```

```
this list contains five words.
```

Python Strings are a type of *container*. Containers are data types that contain other data. A container is an object with complex properties. Whereas a variable has a name -say `a`- and a single value is stored in it -say `2`-, containers contains multitudes of values and often (as we will see later) can contain variables of different type (`int` and `bool` etc).

```
[2]: mystring = 'this list contains forty one characters!!'  
print(mystring)  
  
len(mystring) # let's check the length of the sting (count the number of_  
↳characters)
```

```
this list contains forty one characters!!
```

```
[2]: 41
```

**Containers** are one of the major ways Python offers to work with data. Later on we will learn about one of the most commonly used Python containers (the [Pandas DataFrame](#)).

As a start and in addition to `str`, below we will learn about some of the basic containers, most notably `lists`, and then mention also tuples, dictionaries, and sets

So, first we will get an overview of the various containers and then we are going to learn how to work with `lists`. Lists are handy! They are the beginning of data management and organization in Python.

### 1.0.2 Python Lists

Lists are containers of indexed, ordered and mutable data. They are variables used to contain multiple elements. Lists are created by writing a set of values inside square brackets:

```
[3]: mylist = [2, 3, 4, 5] # this list contains 4 numbers
      print(mylist)
```

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

Python lists can contain elements of any type

```
[4]: list_of_int = [10, 4, 2, 5] # This list
      ↪contains integers
      print(list_of_int)
      type(list_of_int)
```

```
[10, 4, 2, 5]
```

```
[4]: list
```

```
[5]: list_of_float = [2.3, 4.3, 5.5, 6.1] # This list
      ↪contains floating numbers
      print(list_of_float)
      type(list_of_float)
```

```
[2.3, 4.3, 5.5, 6.1]
```

```
[5]: list
```

```
[6]: list_of_complex = [2 + 1j, 4 + 2j, 5 + 33j, 9 + 6j] # This list
      ↪contains complex numbers
      print(list_of_complex)
      type(list_of_complex)
```

```
[(2+1j), (4+2j), (5+33j), (9+6j)]
```

```
[6]: list
```

```
[7]: list_of_boolean = [True, False, True, True] # This list
      ↪contains boolean values
      print(list_of_boolean)
      type(list_of_boolean)
```

```
[True, False, True, True]
```

```
[7]: list
```

```
[8]: list_of_strings = ['this', 'is', 'a', 'list', 'of', 'strings'] # This list
      ↪contains strings
      print(list_of_strings)
      type(list_of_strings)
```

```
['this', 'is', 'a', 'list', 'of', 'strings']
```

```
[8]: list
```

Answer the following questions.

- Can you create a list with different types of variable in it? [Use the cell below to create a list containing different types of variables and the following one to describe the result of the experiment in your own words]

```
[9]: my_list = ['cat', 8, 8 + 8j, 8.8, True]
      print(my_list)
      type(my_list)
```

```
['cat', 8, (8+8j), 8.8, True]
```

```
[9]: list
```

[Describe the results of the experiment above here: The experiment works and the list I created contains strings, boolean values, complex and floating numbers, and integers.]

### 1.0.3 Python Tuple

Tuples are ordered collections of data. They are similar to lists but immutable. Whereas you can add new elements to a previously defined lists you cannot do that with Tuples.

Tuples are defined with parenthesis:

```
[10]: mytuple = (9,4,5)
      print(mytuple)
      type(mytuple)
```

```
(9, 4, 5)
```

```
[10]: tuple
```

Answer the following questions.

- Can you create a Tuple with different types of variable in it? [Use the cell below to create a list containing different types of variables and the following one to describe the result of the experiment in your own words]

```
[13]: my_tuple = (True, 8.8, 8, "dog")
      print(my_tuple)
      type(my_tuple)
```

```
(True, 8.8, 8, 'dog')
```

[13]: tuple

[Describe the results of the experiment above here: Yes you can create tuple with different types of elements. The one I created contains boolean values, strings, floating numbers and integers.]

Lists and Tuple differ in a variety of ways. The most noticeable one is that lists are mutable, tuple are immutable. This means that, we can add elements to a list but not a tuple (in the example we will use the method `.append` to attempt to add elements to a list and a tuple):

```
[14]: mylist = [1,2,3] # this list has three elements
      print(mylist)
      type(mylist)
```

```
[1, 2, 3]
```

[14]: list

Now, we attempt to add a forth element to the list (using `.append()`)

```
[15]: mylist.append(4)
      print(mylist)
      type(mylist)
```

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

[15]: list

```
[16]: mytuple = (1,2,3)
      print(mytuple)
      type(mytuple)
```

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

[16]: tuple

Now, we attempt to add a forth element to the tuple

```
[17]: mytuple.append(4) # it does not work
```

```

-----
AttributeError                                Traceback (most recent call last)
Input In [17], in <cell line: 1>()
----> 1 mytuple.append(4)

AttributeError: 'tuple' object has no attribute 'append'

```

*note. You can explore all the other methods of a container (or any variable in Python) by typing: `containerName.` and then hit the TAB key on your keyboard.*

Answer the following questions.

- How many methods does a Tuple have? [2]
- Which methods does a List have? [append, clear, copy, count, extend, index, insert, pop, remove, reverse, sort]

#### 1.0.4 Python Set

A set is defined as an unordered, unindexed and immutable collection of items. Whereas lists are defined by [], tuple are defined by (), sets are defined by {}.

```

[2]: myset = {"A", "B", "C", "D"}
      print(myset)
      type(myset)

```

```
{'A', 'D', 'C', 'B'}
```

```
[2]: set
```

Unordered means that items in the set do not have an assigned order, so they cannot be indexed:

```
[3]: myset{1}
```

```

Input In [3]
  myset{1}
    ^
SyntaxError: invalid syntax

```

This is means also that a set is immutable and elements cannot be replaced:

```
[4]: myset{2} = "F"
```

```

Input In [4]
  myset{2} = "F"
    ^

```

```
SyntaxError: invalid syntax
```

This is different from Tuple and Lists:

```
[22]: mylist = ["A","B","C","D"]  
      print(mylist[2])
```

C

```
[23]: mylist[3] = "F"  
      print(mylist)
```

```
['A', 'B', 'C', 'F']
```

Answer the following questions.

- In the line above I changed the element of `mylist` indexed by the number 3, why did that operation change the last element in the list? [Report your answer here]

Sets have some cool properties, similar to the “sets” you might have studied in highschool or college. We can perform union operations with sets.

Ok so let’s remind ourselves what `myset` contained:

```
[5]: print(myset)
```

```
{'A', 'D', 'C', 'B'}
```

Now let’s create a new set (`myset2`) with numbers in it, and then construct the union of `myset` and `myset2`, we will save the union set in a variable called `myset3`:

```
[25]: myset2 = {'E', 'F', 'G'}  
      print(myset2)  
      myset3 = {}  
      myset3 = myset.union(myset2)  
      print(myset3)
```

```
{'F', 'E', 'G'}
```

```
{'F', 'E', 'D', 'C', 'B', 'A', 'G'}
```

Answer the following questions.

- can you repeat the same union experiment with lists? For example, can you take the list `mylist` create a second one called `mylist2` containing other elements (your pick) and create a third list `mylist3` that is the union of `mylist` and `mylist2`?

[Use the cell below to try the experiment]

[The union of the lists does not work. There is no function for it.]

```
[27]: mylist2 = [1, 2, 3]
      mylist3 = []
      mylist3 = mylist.union(mylist2)
```

```
-----
AttributeError                                Traceback (most recent call last)
Input In [27], in <cell line: 3>()
      1 mylist2 = [1, 2, 3]
      2 mylist3 = []
----> 3 mylist3 = mylist.union(mylist2)

AttributeError: 'list' object has no attribute 'union'
```

Answer the following questions.

- can lists or sets contain a mixture of different data types, say letters, float numbers and integers? Attempt to make a list containing letters, float numbers and integers call it `myWonderList`, show its result using `print()`, then also make a set containing letters, float numbers and integers, call it `myWonderSet`, show its result using `print()`:

[Use the cell below to implement the experiment]

[Both my set and my list were able to have different data types (floating numbers, integers, and strings).]

```
[12]: myWonderList = ['a', 8.8, 8]
      print(myWonderList)

      myWonderSet = {'c', 8.8, 6}
      print(myWonderSet)
```

```
['a', 8.8, 8]
{8.8, 'c', 6}
```

### 1.0.5 Python Dictionaries

A dictionary is an ordered, changeable, collection of items but do not allow duplicates. Dictionaries are defined by pairs of variables where one of the two variables in the pair is generally thought of as a label, the other one as a value. Whereas lists are defined by `[]`, tuple are defined by `()`, dictionaries just like sets are defined by `{}`.

The structure of the content of dictionaries is differently organized than, lists, tuples and sets:

```
[13]: mydictionary = {
      "apples": 3,
      "oranges": 5,
      "bananas": 2,
      "pinapples": 1}
      print(mydictionary)
```

```
type(mydictionary)
```

```
{'apples': 3, 'oranges': 5, 'bananas': 2, 'pinapples': 1}
```

```
[13]: dict
```

Dictionaries allow efficient and human friendly addressing of items. For example, we can recall how many oranges we have in the dictionary by directly calling the oranges label and the corresponding value in the pair of values will be returned:

```
[14]: mydictionary["oranges"]
```

```
[14]: 5
```

Answer the following questions.

- Use the cell below to write code that let you find out how many bananas we have in our container mydictionary of fruits? [2]

```
[15]: mydictionary["bananas"]
```

```
[15]: 2
```

### 1.0.6 A few exercises as a reminder

Answer the following questions.

- Build a List of pets called `home`, where each pet name (a string) is followed by the number of pets I own and have at home (say (cats, 2, dogs, 1, etc)?)

[Show your list in the cell below]

```
[17]: home = ['cat', 2, 'dog', 1, 'bird', 3]
      print(home)
```

```
['cat', 2, 'dog', 1, 'bird', 3]
```

- Build a dictionary called `home_dict`, similar to the `home` list but using the features of the dictionary where a pet name can be associated to its corresponding number.

[Show your dictionary in the cell below]

```
[18]: home_dict = {
      "cats": 2,
      "dog": 1,
      "bird": 3}
      print(home_dict)
      type(home_dict)
```

```
{'cats': 2, 'dog': 1, 'bird': 3}
```



```
[18]: dict
```

- Find the number of dogs in `home` and in `home_dict`.

[Show your code in the cell below]

```
[19]: home_dict["dog"]
```

```
[19]: 1
```

[Use this cell, to explain in your own words how you found the dog and the corresponding number of dogs in `home` first and in `home_dict` after that]

### 1.0.7 More practicing with using Python Lists

Although Lists, Containers, Tuple, and Dictionary are all interesting and important datatypes in python, we will use primarily Lists in this class. Below some additional material to practice making lists and accessing them.

Lists in Python are just what they sound like, lists of things. We make them using [square brackets].

```
[20]: mylist = [1, 3, 5, 7, 11, 13]
```

A list is an extension of a regular (or “scalar”) **variable**, which can only hold one thing at the time.

```
[21]: notalist = 3.14
```

```
[22]: print(notalist)
```

```
3.14
```

```
[23]: print(mylist)
```

```
[1, 3, 5, 7, 11, 13]
```

Aside: If we’re working with only numbers, then you can think of a regular variable as a “scalar” and a list as a “vector”.

Lists can, however, hold things besides numbers. For example, they can hold ‘text’.

```
[24]: mylist2 = ['this', 'is', 'a', 'list', 'of', 'words']
```

```
[25]: mylist2
```

```
[25]: ['this', 'is', 'a', 'list', 'of', 'words']
```

(Some people, even us, might casually call this a vector but that’s technically not true.)

In reality, lists can hold all sort of things, say numbers (scalars), ‘text’ and even other lists, and all at once.

```
[26]: mylist3 = [1, 'one', [2, 3, 4]]
```

```
[27]: mylist3
```

```
[27]: [1, 'one', [2, 3, 4]]
```

Note that this last list holds a list at index=2

We can get elements of a list by using **index** values in square brackets.

```
[28]: mylist
```

```
[28]: [1, 3, 5, 7, 11, 13]
```

```
[29]: mylist[5]
```

```
[29]: 13
```

```
[30]: mylist3[2]
```

```
[30]: [2, 3, 4]
```

**Python uses a 0-based indexing, not a 1-based indexing (the first value in container is indexed with the number 0). This means that the first value in a list is at index=0, not index=1. This is different than many other languages including R and MatLab!**

We can address more than one element in a list by using the : (colon) operator.

```
[31]: mylist[0:3]
```

```
[31]: [1, 3, 5]
```

We can read this as “Give me all the elements in the interval between 0 **inclusive** to 3 **exclusive**.”

I know this is weird. But at least for any two indexes **a** and **b**, the number of elements you get back from `mylist[a,b]` is always equal to **b** minus **a**, so I guess that’s good!

We can get any consecutive hunk of elements using `:`.

```
[32]: mylist[2:5]
```

```
[32]: [5, 7, 11]
```

If you omit the indexes, Python will assume you want everything.

```
[33]: mylist[:]
```

```
[33]: [1, 3, 5, 7, 11, 13]
```

That doesn’t seem very useful... But, actually, it will turn out to be **really** useful later on, when we will start using numpy arrays!

If you just use one index, the `:` is assumed to mean “from the beginning” or “to the end”. Like this:

```
[34]: mylist[:3] # from the beginning to 3
```

```
[34]: [1, 3, 5]
```

And this:

```
[35]: mylist[3:] # from 3 to the end
```

```
[35]: [7, 11, 13]
```

In addition to the `list[start:stop]` syntax, you can add a step after a second colon, as in `list[start:stop:step]`. This asks for all the element between `start` and `stop` but in steps of `step`, not necessarily consecutive elements. For example every other element:

```
[36]: mylist[0:5:2] # get every other element
```

```
[36]: [1, 5, 11]
```

As you’ve probably figured out, all our outputs above have been lists. So if we assign the output a name, it will be another list.

```
[38]: every_other_one = mylist[0:-1:2] # could also do mylist[0::2]
```

```
[39]: every_other_one
```

```
[39]: [1, 5, 11]
```

See!

If we want a group of elements that aren’t evenly spaced, we’ll need to specify the indexes “by hand”.

```
[40]: anothernewlist = [mylist[1],mylist[2],mylist[4]]
```

```
[41]: anothernewlist
```

```
[41]: [3, 5, 11]
```

So those are the basics of lists. They:

- store a list of things (duh)
- start at index zero
- can be accessed using three things together:
  - square brackets `[]`
  - integer indexes (including negative “start from the end” indexes)
  - a colon `:` (or two if you want a step value other than 1)

```
[42]: letters = [2, 4, 3, 6, 5, 8, 11, 10, 9, 14, 13, 12, 7]
      # 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'

      string_letters = str(letters)
      lists_letters = list(letters)
      tuples_letters = tuple(letters)
      sets_letters = set(letters)

      print("String: ", string_letters)
      print("Lists: ", lists_letters)
      print("Tuples: ", tuples_letters)
      print("Sets: ", sets_letters)
```

```
String: [2, 4, 3, 6, 5, 8, 11, 10, 9, 14, 13, 12, 7]
Lists:  [2, 4, 3, 6, 5, 8, 11, 10, 9, 14, 13, 12, 7]
Tuples: (2, 4, 3, 6, 5, 8, 11, 10, 9, 14, 13, 12, 7)
Sets:   {2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14}
```

```
[ ]:
```