Introduction to Advanced Python Data Types

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 turorial, `str` (strings) are a little bit different. They can store multiple characters all at onces. For example:

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

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

Python Strings are a type of *container* (https://en.wikipedia.org/wiki/Container_(abstract_data_type)). 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).

In [2]:
```
mystring = 'this list contains forty one characters!!'
print(mystring)

len(mystring) # Let's check the length of the sting (count the number of characte
```

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

Out[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 (https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html)).

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! Thye are the beginning of data management and organization in Python.

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 brakets:

In [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

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

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

Out[4]:
```
list
```

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

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

Out[5]:
```
list
```

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

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

Out[6]:
```
list
```

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

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

Out[7]:
```
list
```

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

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

Out[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]

In [9]:
```
list_different_vars = [1, 'string', 2.1]                    # list w/ different
print(list_different_vars)
print("list type: ", type(list_different_vars))
print("1st element type: ", type(list_different_vars[0]))
```

```
[1, 'string', 2.1]
list type:  <class 'list'>
1st element type:  <class 'int'>
```

Yes, a list can contain different types of variables in it. The variable that stores the list is still called a "list" type variable, and each element in the list may be of a different type.

Python Tuple

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

Tuples are defined with parenthesis:

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

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

Out[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]

In [11]:
```
tuple_different_vars = (1.1, 'string', 2)                    # tuple w/ differer
print(tuple_different_vars)
print("list type: ", type(tuple_different_vars))
print("1st element type: ", type(tuple_different_vars[0]))
```

```
(1.1, 'string', 2)
list type:  <class 'tuple'>
1st element type:  <class 'float'>
```

Yes, a tuple can contain different types of variables in it. The variable that stores the tuple is still called a "tuple" type variable, and each element in the tuple may be of a different type.

Lists and Tuple differ in a variety of ways. The most noticble 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):

In [12]:
```
mylist = [1,2,3] # this list has three elements
print(mylist)
type(mylist)
```

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

Out[12]:
```
list
```

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

In [13]:
```
mylist.append(4)
print(mylist)
type(mylist)
```

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

Out[13]:
```
list
```

In [14]:
```
mytuple = (1,2,3)
print(mytuple)
type(mytuple)
```

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

Out[14]:
```
tuple
```

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

In [15]:
```
mytuple.append(4) # it does not work
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Input In [15], 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?

  1. append()
  2. clear()
  3. copy()
  4. count()
  5. extend()
  6. index()
  7. insert()
  8. pop()
  9. remove()
  10. reverse()
  11. sort()

Python Set

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

In [16]:
```
myset = {"A", "B", "C", "D"}
print(myset)
type(myset)
```

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

Out[16]:
```
set
```

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

In [17]:
```
myset{1} # error
```

```
  Input In [17]
    myset{1} # error
         ^
SyntaxError: invalid syntax
```

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

In [18]:
```
myset{2} = "F" # error
```

```
  Input In [18]
    myset{2} = "F" # error
         ^
SyntaxError: invalid syntax
```

This is different from Tuple and Lists:

In [19]:
```
mylist = ["A","B","C","D"]
print(mylist[2])
```

```
C
```

In [20]:
```
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?

  List use a 0-based index, meaning that the first element is located at index of 0, the next element is located at index of 1, etc. Thus, the last element in the list (i.e. 4th element) is located at index of 3, and this index can be used to access and change the last element.

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:

In [21]:
```
print(myset)
```

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

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` :

In [22]:
```
myset2 = {'E', 'F', 'G'}
print(myset2)
myset3 = {'A'}
myset3 = myset.union(myset2)
print(myset3)
```

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

Answer the following questions.

- can you repeate 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]

  No, you cannot repeat the same union experiment with lists. The List class does not have union() function.

```
In [1]: # create two lists
        mylist = ['E', 'F', 'G']
        mylist2= ['A', 'B', 'C']
        print("mylist1: ", mylist)
        print("mylist2: ", mylist2)

        # try creating union of two lists
        mylist3 = mylist.union(mylist2) # error
        print(mylist3)
```

```
mylist1:  ['E', 'F', 'G']
mylist2:  ['A', 'B', 'C']

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Input In [1], in <cell line: 8>()
      5 print("mylist2: ", mylist2)
      7 # try creating union of two lists
----> 8 mylist3 = mylist.union(mylist2) # error
      9 print(mylist3)

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]

Yes, both lists and sets can contain a mixture of different data types. Interesting to note is that the list stores the elements in order and prints it in the order while the set is unordered, so it doesn't print in the same order as when you initialized it (it prints it out in chronological and alphabetically order).

In [24]:
```
myWonderList = [1, 'a',  2.1]
print(myWonderList)

myWonderSet = {3, 'b', 4.5}
print(myWonderSet)
```

```
[1, 'a', 2.1]
{3, 4.5, 'b'}
```

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 liek sets are defined by `{}` .

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

In [25]:
```
mydictionary = {
    "apples": 3,
    "oranges": 5,
    "bananas": 2,
    "pinapples": 1}
print(mydictionary)
type(mydictionary)
```

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

Out[25]:
```
dict
```

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

In [26]:
```
mydictionary["oranges"]
```

Out[26]:
```
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

In [27]:
```
mydictionary["bananas"]
```

Out[27]:
```
2
```

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]

In [28]:
```
home = ["cats", 2, "dogs", 1]
print(home)
```

```
['cats', 2, 'dogs', 1]
```

- 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 correspoding number.

  [Show your dictionary in the cell below]

In [29]:
```
home_dict = {
    "cats": 2,
    "dogs": 1}
print(home_dict)
```

```
{'cats': 2, 'dogs': 1}
```

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

  [Show your code in the cell below]

In [30]:
```
print("home list: ", home[3])           # list
print("home_dict: ", home_dict["dogs"])   # dictionary
```

```
home list:  1
home_dict:  1
```

To find the number of dogs in `home`, I used the index. Since I stored the number of dogs in the 4th element and it's a list, I used square brackets and the index `[3]` to print out the answer. In contrast, I used the label to locate the value (i.e. number of dogs) in `home_dict`. I used square brackets and the label `["dogs"]` to print out the answer.

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 thsi 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]`.

In [31]:
```
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.

In [32]:
```
notalist = 3.14
```

In [33]:
```
print(notalist)
```
```
3.14
```

In [34]:
```
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'.

In [35]:
```
mylist2 = ['this', 'is', 'a', 'list', 'of', 'words']
```

In [36]:
```
mylist2
```
Out[36]:
```
['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.

In [37]:
```
mylist3 = [1, 'one', [2, 3, 4]]
```

In [38]:
```
mylist3
```
Out[38]:
```
[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.

In [39]:
```
mylist
```
Out[39]:
```
[1, 3, 5, 7, 11, 13]
```

In [40]:
```
mylist[5]
```
Out[40]:
```
13
```

In [41]:
```
mylist3[2]
```
Out[41]:
```
[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.

In [42]:
```
mylist[0:3]
```
Out[42]:
```
[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 `:`.

In [43]:
```
mylist[2:5]
```
Out[43]:
```
[5, 7, 11]
```

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

In [44]:
```
mylist[:]
```
Out[44]:
```
[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:

In [45]:
```
mylist[:3] # from the beginning to 3
```
Out[45]:
```
[1, 3, 5]
```

And this:

In [46]:
```
mylist[3:] # from 3 to the end
```
Out[46]:
```
[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:

In [47]:
```
mylist[0:5:2] # get every other element
```
Out[47]:
```
[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.

In [48]:
```
every_other_one = mylist[0:-1:2] # could also do mylist[0::2]
```

In [49]:
```
every_other_one
```
Out[49]:
```
[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".

In [50]:
```
anothernewlist = [mylist[1],mylist[2],mylist[4]]
```

In [51]:
```
anothernewlist
```
Out[51]:
```
[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)

In [52]:
```
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}
```

In [ ]: