3_datatypes

October 17, 2023

1 Data Types in Python

It's time to look at different data types we may find useful in our course. Besides the number types mentioned previously, there are also other types like **strings**, **lists**, **tuples**, **dictionaries** and **sets**.

Data Types	Classes	Description
Numeric	int, float, complex	holds numeric values
String	str	stores sequence of characters
Sequence	list, tuple, range	stores collection of items
Mapping	dict	stores data as a key-value:pair
Boolean	bool	holds either True or False
Set	set, frozeenset	hold collection of unique items

Each of these data types has a number of connected `methods` (functions) which allow to manipulate the data contained in a variable. If you want to know which methods are available for a certain object use the command `dir`, e.g.

```
s="string"
dir(s)
```

The following few cells will give you a short introduction into each type.

1.1 Strings

Strings are lists of keyboard characters as well as other characters not on your keyboard. They are useful for printing results on the screen, during reading and writing of data.

[59]:
```
s="Hello" # string variable
type(s)
```

[59]:
```
str
```

[60]:
```
t="world!"
```

String can be concatenated using the + operator.

[66]:
```
c=s+' '+t
```

[67]:
```
print(c)
```

```
Hello world!
```

As strings are lists, each character in a string can be accessed by addressing the position in the string (see Lists section)

[70]:
```
c[0]
```

[70]:
```
'H'
```

Strings can also be made out of numbers.

[71]:
```
"975"+"321"
```

[71]:
```
'975321'
```

If you want to obtain a number of a string, you can use what is known as type casting. Using type casting you may convert the string or any other data type into a different type if this is possible. To find out if a string is a pure number you may use the `str.isnumeric` method. For the above string, we may want to do a conversion to the type *int* by typing:

[72]:
```
("975"+"321").isnumeric() # or you may use as well str.isnumeric("975"+"321")
```

[72]:
```
True
```

[73]:
```
int("975"+"321")
```

[73]:
```
975321
```

There are a number of methods connected to the string data type. Usually the relate to formatting or finding sub-strings. Formatting will be a topic in our next lecture. Here we just refer to one simple find example.

[74]:
```
t.find('ld') ## returns the index at which the sub string 'ld' starts in t
```

[74]:
```
3
```

[76]:
```
t.capitalize()
```

[76]:
```
'World!'
```

1.2 Lists

Lists have a variety of uses. They are useful, for example, in various bookkeeping tasks that arise in computer programming. Like arrays, they are sometimes used to store data. However, lists do not have the specialized properties and tools that make arrays so powerful for scientific computing. So in general, we prefer arrays to lists for working with scientific data. For other tasks, lists work just fine and can even be preferable to arrays.

```
[78]: a=[]
```

```
[80]: a = [0, 1, 1, 2, 3, 5, 8, 13]
```

```
[81]: b = [5., "girl", 2+0j, "horse", 21]
```

Individual elements in a list can be accessed by the variable name and the number (index) of the list element put in square brackets. Note that the index for the elements start at *0* for the first element (left).

Note Indices in Python

The first element of a list or array is accessed by the index 0. If the array has N elements, the last entries index is N-1.

```
[85]: b[0],b[1]
```

```
[85]: (5.0, 'girl')
```

Elements may be also accessed from the back by nagative indices. *b[-1]* denotes the last element in the list and *b[-2]*, the element before the last.

```
[86]: b[-1]
```

```
[86]: 21
```

```
[87]: b[-2]
```

```
[87]: 'horse'
```

The length of a list can be obtained by the `len` command if you need the number of elements in the list for your calculations.

```
[88]: len(b)
```

```
[88]: 5
```

There are powerful ways to iterate through a list and also through arrays in form of *iterator*. This is called *list comprehension*. We will talk about them later in more detail. Here is an example, which shows the powerful options you have in Python.

```
[89]: [x+2 for x in b if type(x)!=str]
```

```
[89]: [7.0, (4+0j), 23]
```

Individual elements in a list can be replaced by assigning a new value to them

```
[90]: b[-2]='cat'
```

```
[91]: b
```

```
[91]: [5.0, 'girl', (2+0j), 'cat', 21]
```

Lists can be concatanated by the + operator

```
[93]: c=a+b
      c
```

```
[93]: [0, 1, 1, 2, 3, 5, 8, 13, 5.0, 'girl', (2+0j), 'cat', 21]
```

A very useful feature for lists in python is the **slicing** of lists. Slicing means, that we access only a range of elements in the list, i.e. element 3 to 7. This is done by inserting the starting and the ending element number separated by a colon (:) in the square brackets. The index numbers can be positive or negative again.

```
[95]: c[3:6]
```

```
[95]: [2, 3, 5]
```

Inserting a second colon behind the ending element index together with a thrid number allows even to select only ever second or third element from a list.

```
[96]: c[3:9:2]
```

```
[96]: [2, 5, 13]
```

It is sometimes also useful to reverse a list. This can be easily done with the reverse command.

```
[97]: c.reverse()
      c
```

```
[97]: [21, 'cat', (2+0j), 'girl', 5.0, 13, 8, 5, 3, 2, 1, 1, 0]
```

Lists may be created in different ways. An empty list can be created by assigning emtpy square brackets to a variable name. You can append elements to the list with the help of the append command which has to be added to the variable name as shown below. This way of adding a particular function, which is part of a certain variable class is part of object oriented programming.

```
[109]: a=[0]
```

```
[110]: a.append(7)
```

```
[98]: a=[10]*10
```

```
[99]: a
```

```
[99]: [10, 10, 10, 10, 10, 10, 10, 10, 10, 10]
```

A list of numbers can be easily created by the `range()` command.

```
[45]: list(range(1,10))
```

```
[45]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
[100]: list(range(3,10))
```

```
[100]: [3, 4, 5, 6, 7, 8, 9]
```

```
[103]: list(range(3,10,2))
```

```
[103]: [3, 5, 7, 9]
```

Lists (and also tuples below) can be multidimensional as well, i.e. for an image. The individual elements may then be addressed by supplying two indices in two square brackets.

```
[104]: a = [[3, 9], [8, 5], [11, 1]]
```

```
[107]: a[1]
```

```
[107]: [8, 5]
```

```
[106]: a[2][0]
```

```
[106]: 11
```

1.3 Tuples

Tuples are also list, but immutable. That means, if a tuple has been once defined, it cannot be changed. Try to change an element to see the result.

```
[113]: point = (10, 20)

       print(point, type(point))
```

```
(10, 20) <class 'tuple'>
```

Tuples may be unpacked, e.g. its values may be assigned to normal variables in the following way

```
[137]: point[1]
```

```
[137]: 20
```

```
[116]: x, y = point

       print("x =", x)
       print("y =", y)
```

```
x = 10
y = 20
```

1.4 Dictionaries

Dictionaries are like lists, but the elements of dictionaries are accessed in a different way than for lists. The elements of lists and arrays are numbered consecutively, and to access an element of a list or an array, you simply refer to the number corresponding to its position in the sequence. The elements of dictionaries are accessed by “keys”, which can be either strings or (arbitrary) integers (in no particular order). Dictionaries are an important part of core Python. However, we do not make much use of them in this introduction to scientific Python, so our discussion of them is limited.

```
[123]: room = {"a":"b", "Frank":322, "Dekan":550}
```

```
[124]: room['a']
```

```
[124]: 'b'
```

```
[120]: room.keys()
```

```
[120]: dict_keys(['Ralf', 'Frank', 'Dekan'])
```

```
[121]: room.values()
```

```
[121]: dict_values([422, 322, 550])
```

1.5 Sets

Sets are like lists but have immutable unique entries, which means the elements can not be changed once defined. An emtpy set is created by the `set()` method.

```
[125]: phone_id = {112, 114, 116, 118, 115}
       print('Phone ID:', phone_id)
```

```
Phone ID: {112, 114, 115, 116, 118}
```

```
[126]: type(phone_id)
```

```
[126]: set
```

```
[139]: fruits = {'Banana', 'Mango', 'Kiwi', 'Strawberry', 'Orange'}
       print('Fruits:', fruits)
```

```
Fruits: {'Strawberry', 'Orange', 'Mango', 'Banana', 'Kiwi'}
```

```
[128]: mixed_set = {'Python', 112, -2, 'Good'}
       print('Set of mixed data types:', mixed_set)
```

```
Set of mixed data types: {112, 'Python', -2, 'Good'}
```

You may add elements to a set with the `add` method:

You may also remove objects with the `discard` method:

```
[24]: fruits.discard("Orange")
```

You may also apply a variety of operation to sets checking their *intersection*, *union* or *difference*.

```
[140]: A = {1, 3, 5}
       B = {1, 2, 3}
```

```
[141]: #intersection
       A & B
```

```
[141]: {1, 3}
```

```
[145]: #symmetric difference
       B^A
```

```
[145]: {2, 5}
```

```
[146]: # union
       A | B
```

```
[146]: {1, 2, 3, 5}
```