python_lecture_02_data_structures

January 29, 2020

1 Programming with Python

2 2 Sequential data types

Sequential data types are **compound data types**, i.e. they are used to group other values together. *Lists* and *Dictionaries* are the two most important data structures in python.

2.1 2.1 Lists

Lists are like strings: they are **ordered sequences** of objects. List literals are sequences of comma-separated values that are enclose in *square* brackets*:

[1]:
```
# create a list of word characters and print it:
my_list = [ 'M', 'o', 'n', 't', 'y']
print(my_list)
```

```
['M', 'o', 'n', 't', 'y']
```

Lists are like strings: an element is accessed by referencing its index:

[2]:
```
first_letter = my_list[0]
print(first_letter)
```

```
M
```

Like strings, lists can also reference **ranges**, exactly the same rules apply:

[3]:
```
my_list[0:3]
```

[3]:
```
['M', 'o', 'n']
```

[4]:
```
my_list[2:]
```

[4]:
```
['n', 't', 'y']
```

Lists are unlike strings: while strings are immutable, **lists are mutable**:

[5]:
```
# this does not work with strings, because they are immutable:
my_string = 'Monty'
my_string[0] = 'm'
print(my_string)
```

```
␣
↪---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call␣
↪last)

<ipython-input-5-885917591556> in <module>
  1 # this does not work with strings, because they are immutable:
  2 my_string = 'Monty'
----> 3 my_string[0] = 'm'
  4 print(my_string)


TypeError: 'str' object does not support item assignment
```

[ ]:
```
# because lists are mutable, this works
my_list = [ 'M', 'o', 'n', 't', 'y']
my_list[0] = 'X'
print(my_list)
```

[6]:
```
# re-assign multiple, consecutive values with ranges
my_list = [ 'M', 'o', 'n', 't', 'y']
my_list[2:4] = ['x', 'x']
my_list
```

[6]:
```
['M', 'o', 'x', 'x', 'y']
```

[7]:
```
# values can be deleted by assigning empty squares brackets to them
my_list[2:4] = []
my_list
```

[7]:
```
['M', 'o', 'y']
```

[8]:
```
# clearing a list
my_list[:] = []
my_list
```

[8]:
```
[]
```

New elements can be **inserted into a list** using the *insert* method:

[9]:
```
my_list = [ 'M', 'o', 'n', 't', 'y']
my_list.insert(3, 'h')
my_list
```

[9]:
```
['M', 'o', 'n', 'h', 't', 'y']
```

In contrast to modification by re-assignment, the original element at the specified index does not get deleted. Instead, the current element at the specified index and all following elements get shifted up by one index position. The new element is then placed on the now empty index position.

However, adding a new item to the *end of a list* with -1 as index does not work:

[10]:
```
my_list = [ 'M', 'o', 'n', 't', 'y']
my_list.insert(-1, 'XYZ')
my_list
```

[10]:
```
['M', 'o', 'n', 't', 'XYZ', 'y']
```

New elements can be added to the *end of a list* using the **append method**:

[11]:
```
# creates a list of characters (size 1-strings):
letters = ['a','b','c','e','f']
print(letters)
```

```
['a', 'b', 'c', 'e', 'f']
```

[12]:
```
# and add a new item at the end of the list:
letters.append('K')
print(letters)
```

```
['a', 'b', 'c', 'e', 'f', 'K']
```

Lists are unlike strings: while strings get copied when assigning them to a new variable, lists get aliased:

[13]:
```
# create a new list named a
a = ['H', 'e', 'l', 'l', 'o']
# reassign my_list to a new identifier
b = a
# and print them both
print('a =', a)
print('b =', b)
```

```
a = ['H', 'e', 'l', 'l', 'o']
b = ['H', 'e', 'l', 'l', 'o']
```

[14]:
```
# now change the first letter of a ...
a[0] = 'x'
# ... and print both lists again
print('a =', a)
print('b =', b)
```

```
a = ['x', 'e', 'l', 'l', 'o']
b = ['x', 'e', 'l', 'l', 'o']
```

Since a and b yield the same result, both identifiers must point to the same in-memory object! Therefore, the statement $b = a$ has not copied a and assigned the newly created copy to b but has created a second reference from b to the same object that a was already referencing.

Lists are unlike strings: while strings can only store characters, **lists can store any type of object**:

[15]:
```
# create a list of integers and print it
my_list = [0,-2, 3, 1]
print(my_list)
```

```
[0, -2, 3, 1]
```

While list *frequently* store only objects of one type, i.e. number-only or character-only lists, basically **they may store mixed data types**:

[16]:
```
# lists can take mixed types of elements,
# e.g. you can mix strings and numbers:
my_list = [0,-2,3,'a']
print(my_list)
```

```
[0, -2, 3, 'a']
```

If you do not just want to delete the last element, but also get the deleted element as a return value, use the **pop method** instead:

[17]:
```
letters = ['a', 'b', 'c', 'e', 'f', 'K']
last_letter = letters.pop()
print(last_letter)
```

```
K
```

Using **append** and **pop**, lists can be used as **stacks**. Stacks are ordered lists that obey the **last in/first out**-principle. New elements get attached to the end of a stack and if an element is to be removed, the last added element gets removed first:

[18]:
```
# define a list called 'stack'
stack = [0,1,2,3]
print('initial: ', stack)
# now, add a new element by appending it
stack.append(4)
# add another to the stack
stack.append(5)
print('after adding two elements:', stack)
```

```
initial:  [0, 1, 2, 3]
after adding two elements: [0, 1, 2, 3, 4, 5]
```

[19]:
```
# Now, retrieve an element.
# Because we want to emulate a stack,
# we use the pop method to get the element that was added to the stack last:
retrieved = stack.pop()
print('retrieved =', retrieved)
print('stack after retrieval =', stack)
```

```
retrieved = 5
stack after retrieval = [0, 1, 2, 3, 4]
```

So far, we used the pop method without argument. In this case, it always removes (and yields) the last element of a list. If we provide pop with an index, the specified element is removed and yielded:

[20]:
```
letters = ['a', 'b', 'c', 'e', 'f']
third_letter = letters.pop(2)
print(third_letter)
```

```
c
```

As you can see, 'c' has actually been removed from the letters list:

[21]:
```
print(letters)
```

```
['a', 'b', 'e', 'f']
```

Especially, we can use *list.pop(0)* to remove and yield the first element of a list:

[22]:
```
letters = ['a', 'b', 'c', 'e', 'f']
first_letter = letters.pop(0)
print('first_letter =', first_letter)
print('letters after pop(0) =', letters)
```

```
first_letter = a
letters after pop(0) = ['b', 'c', 'e', 'f']
```

Using append and pop(0), lists can be used as **queues**. **Queues** are ordered lists that obey the **first in/first out**-principle. New elements get attached to the end of a queue but if an element is to be removed, the earliest added element gets removed first:

[23]:
```
# define a list called 'queue'
queue = [0,1,2,3]
print('initial: ', queue)
# now, add a new element by appending it
queue.append(4)
# add another to the stack
queue.append(5)
print('after adding two elements:', queue)
```

```
initial:  [0, 1, 2, 3]
after adding two elements: [0, 1, 2, 3, 4, 5]
```

[24]:
```
# Now, retrieve an element.
# Because we want to emulate a queue,
# we use the pop(0) method to get the element that was added to the stack first:
retrieved = queue.pop(0)
print('retrieved =', retrieved)
```

```
print('queue after retrieval =', queue)
```

```
retrieved = 0
queue after retrieval = [1, 2, 3, 4, 5]
```

It is possible to create **nested lists**:

[25]:
```
time = [1, 2, 3, 4, 5]
cofs = [0.17, 0.18, 0.15, 0.16, 0.17]
data = [time, cofs]
data
```

[25]:
```
[[1, 2, 3, 4, 5], [0.17, 0.18, 0.15, 0.16, 0.17]]
```

We can access the individual columns of our *data* list by referencing their indices:

[26]:
```
# get time col
t = data[0]
print(t)
```

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

It is easy to get the cof at 4 seconds like this:

[27]:
```
# get index of element 4 by using the index method
j = data[0].index(4)
print('j =', j)
# now, get the cof at index j from col 2 (index i=1)
cof = data[1][j]
print('cof =', cof)
```

```
j = 3
cof = 0.16
```

Lists can be **concatenated** by using the **plus operator**:

[28]:
```
a = [0,-2,3,'a']
b = [98, 99, 100]
a + b
```

[28]:
```
[0, -2, 3, 'a', 98, 99, 100]
```

[29]:
```
# however, in memory, a still is the same!
print(a)
```

```
[0, -2, 3, 'a']
```

[30]:
```
# change a by reassigning the result of the + operation
a = a + b
# now list a has been changed in memory
print(a)
```

```
[0, -2, 3, 'a', 98, 99, 100]
```

some useful functions for numbers-only lists (integers, floats)

[31]:
```
# get the number of objects in your list by using the built-in function len()
numbers = [0,2.0,4,-3]
len(numbers)
```

[31]:
```
4
```

[32]:
```
# get the sum of all list elements
sum(numbers)
```

[32]:
```
3.0
```

[33]:
```
# get the highest value
max(numbers)
```

[33]:
```
4
```

[34]:
```
# get the lowest value
min(numbers)
```

[34]:
```
-3
```

[35]:
```
# sort list by ascending values
sorted(numbers)
```

[35]:
```
[-3, 0, 2.0, 4]
```

3 2.2 Tuples

A tuple is a data structure very similar to the list data structure. The main difference being that tuple manipulation are faster than list because tuples are **immutable**.

To create a tuple, place values within **round** brackets:

[36]:
```
t = (1,2,3)
print(t)
```

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

It is also possible to create a tuple without parentheses, by using commas:

[37]:
```
t2 = 4,5,6
print(t2)
```

```
(4, 5, 6)
```

If you want to create a tuple with a single element, you must use the comma:

[38]:
```
single = (1, )
print(single)
```

```
(1,)
```

You can repeat a tuples by multiplying a tuple by a number:

[39]:
```
t3 = single * 8
print(t3)
```

```
(1, 1, 1, 1, 1, 1, 1, 1)
```

Access tuple is identical to lists:

[40]:
```
print(t[1])
```

```
2
```

4 2.3 Dictionaries

A dictionary is a sequence of items. Each item is a pair made of a **key** and a **value**. Dictionaries are **not sorted**. You can access to the list of keys or values independently.

[41]:
```
car = {'brand' : 'Porsche', 'model' : 'Cayenne', 'price': 76_690}
print(car)
```

```
{'brand': 'Porsche', 'model': 'Cayenne', 'price': 76690}
```

How to display all keys and values:

[42]:
```
print(car.keys())
```

```
dict_keys(['brand', 'model', 'price'])
```

[43]:
```
print(car.values())
```

```
dict_values(['Porsche', 'Cayenne', 76690])
```

[44]:
```
print(car.items())
```

```
dict_items([('brand', 'Porsche'), ('model', 'Cayenne'), ('price', 76690)])
```

How to access values trough keys:

[45]:
```
print(car["brand"])
```

```
Porsche
```

combining dictionaries:

[46]:
```
color = {'color': 'black'}
```

[47]:
```
car.update(color)
print(car)
```

```
{'brand': 'Porsche', 'model': 'Cayenne', 'price': 76690, 'color': 'black'}
```

zip function
The zip function can be used to create a dictionary from list:

[48]:
```
keys = ['brand', 'model', 'price']
values = ['Porsche', 'Cayenne', 76_690]
print('Keys:', keys)
print('Values:', values)
print('Dict:', dict(zip(keys, values)))
```

```
Keys: ['brand', 'model', 'price']
Values: ['Porsche', 'Cayenne', 76690]
Dict: {'brand': 'Porsche', 'model': 'Cayenne', 'price': 76690}
```

[49]:
```
# values can be nested lists as well
keys = ['brand', 'model', 'price']
values = [[1,2,3,4], [4,5,6], [6,78,9], [78,12,123]]
print('Keys:', keys)
print('Values:', values)
print('Dict:', dict(zip(keys, values)))
```

```
Keys: ['brand', 'model', 'price']
Values: [[1, 2, 3, 4], [4, 5, 6], [6, 78, 9], [78, 12, 123]]
Dict: {'brand': [1, 2, 3, 4], 'model': [4, 5, 6], 'price': [6, 78, 9]}
```

[50]:
```
# It also works for lists
print(list(zip(keys, values)))
```

```
[('brand', [1, 2, 3, 4]), ('model', [4, 5, 6]), ('price', [6, 78, 9])]
```

5 2.4 Sets

Sets are constructed from a sequence (or some other iterable object). Sets have **no duplicates**, there are usually used to build sequence of unique items (e.g., set of identifiers).

[51]:
```
a = set([1, 2, 3, 4, 5, 6])
b = set([4, 5, 6, 7, 8, 9])
print('a:', a)
print('b:', b)
```

```
a: {1, 2, 3, 4, 5, 6}
b: {4, 5, 6, 7, 8, 9}
```

[52]:
```
# Union
a | b
```

[52]:
```
{1, 2, 3, 4, 5, 6, 7, 8, 9}
```

[53]:
```
# Intersection
a & b
```

[53]:
```
{4, 5, 6}
```

[54]:
```
# Subset
a < b
```

[54]:
```
False
```

[55]:
```
# Difference
a - b
```

[55]:
```
{1, 2, 3}
```

[56]:
```
# Symmetric difference
a ^ b
```

[56]:
```
{1, 2, 3, 7, 8, 9}
```

Sets are **mutable**:

[57]:
```
a.add('new_value')
print(a)
```

```
{1, 2, 3, 4, 5, 6, 'new_value'}
```

6 2.5 Frozensets

Frozensets are like sets, but immutable, i.e. values cannot be changed or added.

[58]:
```
some_numbers = (1,2,3,4,5)
normalset = set(some_numbers)
frozen_set = frozenset(some_numbers)
print("normalset:", normalset)
print("frozenset:", frozen_set)
```

```
normalset: {1, 2, 3, 4, 5}
frozenset: frozenset({1, 2, 3, 4, 5})
```

[59]:
```
new_value = 42
normalset.add(new_value)
print(normalset)
```

```
{1, 2, 3, 4, 5, 42}
```

[60]:
```
frozenset.add(new_value)
print(frozen_set)
```

```
␣
↪---------------------------------------------------------------------------
```

```
AttributeError                            Traceback (most recent call␣
↪last)

<ipython-input-60-13d8f6988ebd> in <module>
----> 1 frozenset.add(new_value)
      2 print(frozen_set)


AttributeError: type object 'frozenset' has no attribute 'add'
```

7 2.6 Typecast von sequential data types

[61]:
```
some_tuple = (1,2,3)
print(some_tuple,'of',type(some_tuple), "is cast into",␣
 ↪list(some_tuple),'of',type(list(some_tuple)))
```

```
(1, 2, 3) of <class 'tuple'> is cast into [1, 2, 3] of <class 'list'>
```

[62]:
```
some_tuple = [('x', 5), ('Y', 10)]
print(some_tuple,'of',type(some_tuple), "is cast into",␣
 ↪dict(some_tuple),'of',type(dict(some_tuple)))
#= dict([('x', 5), ('y', -5)])
#print('numbers1 =',numbers1)
```

```
[('x', 5), ('Y', 10)] of <class 'list'> is cast into {'x': 5, 'Y': 10} of <class
'dict'>
```

8 2.7 Summary

Type	delimiter	mutable	sorted
list	[ ]	yes	yes
tuple	( )	no	yes
dictionary	{ }	yes	no
set	set( )	yes	no
frozenset	frozenset( )	no	no