03-structures

September 16, 2022

1 Basic containers

1.1 Tuples (immutable)

A tuple is an immutable sequence of arbitrary Python objects.

An empty tuple is defined as

[1]:
```
t = ()
t
```

[1]:
```
()
```

[2]:
```
type(t)
```

[2]:
```
tuple
```

To define a tuple with one single element a comma is needed

[3]:
```
one_element_tuple = (42, )
one_element_tuple
```

[3]:
```
(42,)
```

But is can have many values

[4]:
```
three_elements_tuple = (1, 3, 5)
three_elements_tuple
```

[4]:
```
(1, 3, 5)
```

Of different types

[5]:
```
mixed_tuple = (1, 1.0, True, "DHML")
mixed_tuple
```

[5]:
```
(1, 1.0, True, 'DHML')
```

And multiples values can be assigned in a single line

[6]:
```
a, b, c = three_elements_tuple
'{} - {} - {}'.format(a, b, c)
```

[6]:
```
'1 - 3 - 5'
```

[7]:
```
a, b, c = 1, 2, 3       # tuple for multiple assignment
'{} - {} - {}'.format(a, b, c)
```

[7]:
```
'1 - 2 - 3'
```

1.1.1 Swapping variable values

By the way, we swap the value of two variables like this

[8]:
```
a = "a"
b = "b"
print('before: {} - {}'.format(a, b))
a, b = b, a
print('after: {} - {}'.format(a, b))
```

```
before: a - b
after: b - a
```

or even more fun

[9]:
```
a, b, c = 1, 2, 3
print('before: {} - {} - {}'.format(a, b, c))
a, b, c = b, c, a
print('after: {} - {} - {}'.format(a, b, c))
```

```
before: 1 - 2 - 3
after: 2 - 3 - 1
```

1.1.2 Operation with tuples

The membership operator `in` can be used with lists, strings, dictionaries, and in general with collection and sequence objects.

[10]:
```
3 in three_elements_tuple
```

[10]:
```
True
```

[11]:
```
9 in three_elements_tuple
```

[11]:
```
False
```

1.1.3 Tuples are immutable

The next line will throw an error…

[12]:
```
three_elements_tuple[0] = 10
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In [12], line 1
----> 1 three_elements_tuple[0] = 10

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

1.2 Lists (mutable)

- Mutable sequences differ from their immutable sisters in that they can be changed after creation.
- There are two *mutable* sequence types in Python: lists and byte arrays.
- Lists are very similar to tuples.
- Lists are commonly used to store collections of homogeneous objects, but there is nothing preventing you to store heterogeneous collections as well.

[13]:
```
empty_list = []
another_empty_list = list()
```

[14]:
```
dir(empty_list)
```

[14]:
```
['__add__',
 '__class__',
 '__class_getitem__',
 '__contains__',
 '__delattr__',
 '__delitem__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__gt__',
 '__hash__',
 '__iadd__',
 '__imul__',
 '__init__',
 '__init_subclass__',
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
 '__mul__',
 '__ne__',
```

```
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__reversed__',
 '__rmul__',
 '__setattr__',
 '__setitem__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 'append',
 'clear',
 'copy',
 'count',
 'extend',
 'index',
 'insert',
 'pop',
 'remove',
 'reverse',
 'sort']
```

We can create a list enumerating elements

[15]:
```
lst = [1, 2, 3, 4, 5, 6, 7]
lst
```

[15]:
```
[1, 2, 3, 4, 5, 6, 7]
```

but there are other ways

[16]:
```
list(range(1, 8))
```

[16]:
```
[1, 2, 3, 4, 5, 6, 7]
```

[17]:
```
lst = [i ** 2 for i in range(0, 8)]  # Python is magic
lst
```

[17]:
```
[0, 1, 4, 9, 16, 25, 36, 49]
```

[18]:
```
list((1,2,3))
```

[18]:
```
[1, 2, 3]
```

[19]:
```
list('Hello')
```

[19]:
```
['H', 'e', 'l', 'l', 'o']
```

[20]:
```
20 * '-*'
```

[20]:
```
'-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*'
```

1.2.1 operations

[21]:
```
a  = [1, 2, 3, 4, 1]
dir(a)
```

[21]:
```
['__add__',
 '__class__',
 '__class_getitem__',
 '__contains__',
 '__delattr__',
 '__delitem__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__gt__',
 '__hash__',
 '__iadd__',
 '__imul__',
 '__init__',
 '__init_subclass__',
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
 '__mul__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__reversed__',
 '__rmul__',
 '__setattr__',
 '__setitem__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 'append',
 'clear',
```

```
 'copy',
 'count',
 'extend',
 'index',
 'insert',
 'pop',
 'remove',
 'reverse',
 'sort']
```

we can append anything at the end

[22]:
```
a.append(13)
a
```

[22]:
```
[1, 2, 3, 4, 1, 13]
```

[23]:
```
a.append([11,22,33])
a
```

[23]:
```
[1, 2, 3, 4, 1, 13, [11, 22, 33]]
```

how many 1's are there in the list?

[24]:
```
a.count(1)
```

[24]:
```
2
```

extend the list by another one (or sequence)

[25]:
```
a.extend([11,22,33])
a
```

[25]:
```
[1, 2, 3, 4, 1, 13, [11, 22, 33], 11, 22, 33]
```

insert 111 at position 0...

[26]:
```
a.insert(0, 111)
a
```

[26]:
```
[111, 1, 2, 3, 4, 1, 13, [11, 22, 33], 11, 22, 33]
```

find the position/index of the first occurrence of an element in a list

[27]:
```
a.index(3)
```

[27]:
```
3
```

As for strings, list are 'sliceable'

[28]:
```
a[3]
```

[28]: 3

[29]:
```
a[:-3]
```

[29]: [111, 1, 2, 3, 4, 1, 13, [11, 22, 33]]

pop (remove and return) last element

[30]:
```
 a.pop()
```

[30]: 33

pop element at position 3

[31]:
```
a.pop(3)
```

[31]: 3

[32]:
```
a
```

[32]: [111, 1, 2, 4, 1, 13, [11, 22, 33], 11, 22]

remove the 1st ocurrence of 111 from the list

[33]:
```
a.remove(111)
a
```

[33]: [1, 2, 4, 1, 13, [11, 22, 33], 11, 22]

reverse the order of the elements in the list

[34]:
```
a.reverse()
a
```

[34]: [22, 11, [11, 22, 33], 13, 1, 4, 2, 1]

[35]:
```
a.remove([11, 22, 33])
```

sort the list

[36]:
```
a.sort()
a
```

[36]: [1, 1, 2, 4, 11, 13, 22]

Remove and return the element at position 0

[37]:
```
a.pop(0)
```

[37]: 1

[38]:
```
a
```

[38]:
```
[1, 2, 4, 11, 13, 22]
```

remove all elements from the list

[39]:
```
a.clear()
a
```

[39]:
```
[]
```

as seen, list can have heterogeneous types

[40]:
```
a = list('hello')    # makes a list from a string
a.append(100)        # append 100, heterogeneous type
a
```

[40]:
```
['h', 'e', 'l', 'l', 'o', 100]
```

[41]:
```
a.append((1, 2, 3))
a
```

[41]:
```
['h', 'e', 'l', 'l', 'o', 100, (1, 2, 3)]
```

[42]:
```
a.extend((1, 2, 3)) # extend using tuple.
a
```

[42]:
```
['h', 'e', 'l', 'l', 'o', 100, (1, 2, 3), 1, 2, 3]
```

[43]:
```
a.append(('...',)  )    # extend using string
a
```

[43]:
```
['h', 'e', 'l', 'l', 'o', 100, (1, 2, 3), 1, 2, 3, ('…',)]
```

Other operations

[44]:
```
a = [1, 3, 5, 7]
```

[45]:
```
min(a)                # minimum value in the list
```

[45]:
```
1
```

[46]:
```
max(a)               # maximum value in the list
```

[46]:
```
7
```

[47]:
```
sum(a)               # sum of all values in the list
```

[47]:
```
16
```

[48]:
```
len(a)               # number of elements in the list
```

[48]:
```
4
```

"+" with list means concatenation

[49]:
```
b = [6, 7, 8, 9]
a + b
```

[49]:
```
[1, 3, 5, 7, 6, 7, 8, 9]
```

The zip function acts like a "zip" between two list

[50]:
```
[ i + j for i, j in zip(a, b)]
```

[50]:
```
[7, 10, 13, 16]
```

[51]:
```
list(zip(a, b))
```

[51]:
```
[(1, 6), (3, 7), (5, 8), (7, 9)]
```

"*" has also a special meaning

[52]:
```
a * 3
```

[52]:
```
[1, 3, 5, 7, 1, 3, 5, 7, 1, 3, 5, 7]
```

[53]:
```
10 * "SLB "
```

[53]:
```
'SLB SLB SLB SLB SLB SLB SLB SLB SLB SLB '
```

1.3 Sets

- Python also provides two set types, `set` and `frozenset`.
- The `set` type is mutable, while `frozenset` is immutable.
- They are unordered collections of immutable objects.

[54]:
```
small_primes = set()    # empty set
small_primes
```

[54]:
```
set()
```

adding one element at a time

[55]:
```
small_primes.add(2)
small_primes.add(3)

small_primes
```

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

and now a lot of 5's!

[56]:
```
small_primes.add(5)
small_primes.add(5)
small_primes.add(5)
small_primes
```

[56]:
```
{2, 3, 5}
```

and one more…

[57]:
```
small_primes.add(1)
small_primes
```

[57]:
```
{1, 2, 3, 5}
```

Look what I've done, 1 is not a prime! so let's remove it…

[58]:
```
small_primes.remove(1)
small_primes
```

[58]:
```
{2, 3, 5}
```

1.3.1 operation

[59]:
```
3 in small_primes
```

[59]:
```
True
```

[60]:
```
4 in small_primes
```

[60]:
```
False
```

[61]:
```
4 not in small_primes
```

[61]:
```
True
```

Other forms of creating set are

[62]:
```
bigger_primes = set([5, 7, 11, 13])
bigger_primes
```

[62]:
```
{5, 7, 11, 13}
```

[63]:
```
bigger_primes = {5, 7, 11, 13, 11, 13, 11, 13, 11, 13, 11, 13, 11, 13}
bigger_primes
```

[63]:
```
{5, 7, 11, 13}
```

union operator |

[64]:
```
small_primes | bigger_primes
```

[64]:
```
{2, 3, 5, 7, 11, 13}
```

intersection operator `&`

[65]:
```
small_primes & bigger_primes
```

[65]:
```
{5}
```

difference operator `-`

[66]:
```
small_primes - bigger_primes
```

[66]:
```
{2, 3}
```

1.3.2 Frozen Sets

As already presented, frozen sets are immutable

[67]:
```
small_primes = frozenset([2, 3, 5, 7])
bigger_primes = frozenset([5, 7, 11])
```

we cannot add to a frozenset

[68]:
```
small_primes.add(11)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In [68], line 1
----> 1 small_primes.add(11)

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

neither we can remove

[69]:
```
small_primes.remove(2)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In [69], line 1
----> 1 small_primes.remove(2)

AttributeError: 'frozenset' object has no attribute 'remove'
```

but we can do other operations such as in intersect, union or difference

[70]:
```
small_primes & bigger_primes
```

[70]:
```
frozenset({5, 7})
```

[71]:
```
small_primes | bigger_primes
```

[71]:
```
frozenset({2, 3, 5, 7, 11})
```

[72]:
```
small_primes - bigger_primes
```

[72]:
```
frozenset({2, 3})
```

1.4 Dictionaries

- Dictionary type is the only standard mapping type, and it is the backbone of every Python object.
- A dictionary maps keys to values
- Keys need to be hashable objects, while values can be of any arbitrary type.
- Dictionaries are mutable objects.

[73]:
```
a = dict(A=1, Z=-1)
b = {'A': 1, 'Z': -1}
c = dict(zip(['A', 'Z'], [1, -1]))
d = dict([('A', 1), ('Z', -1)])
e = dict({'Z': -1, 'A': 1})
```

are they equal?

[74]:
```
a == b == c == d == e
```

[74]:
```
True
```

are they all the same?

[75]:
```
[id(x) for x in [a, b, c, d, e]]
```

[75]:
```
[1935294078400, 1935270953088, 1935292754752, 1935293869056, 1935293591360]
```

So, the key can be an immutable and, e.g., tuples are immutables

[76]:
```
B = [42, "Brian"]
a[(1, 2)] = B # (1, 2) is a tuple, so it is immutable
a[(2, 1)] = B
a
```

[76]:
```
{'A': 1, 'Z': -1, (1, 2): [42, 'Brian'], (2, 1): [42, 'Brian']}
```

such as are string

[77]:
```
a["some key"] = 42
```

In `a` was stored a reference to `B`. So if we change what is referenced by `B`

[78]:
```
B.append("life")
a
```

[78]:
```
{'A': 1,
 'Z': -1,
 (1, 2): [42, 'Brian', 'life'],
 (2, 1): [42, 'Brian', 'life'],
 'some key': 42}
```

and more...

[79]:
```
a[1] = 3
a[frozenset([1, 2])] = 12
a
```

[79]:
```
{'A': 1,
 'Z': -1,
 (1, 2): [42, 'Brian', 'life'],
 (2, 1): [42, 'Brian', 'life'],
 'some key': 42,
 1: 3,
 frozenset({1, 2}): 12}
```

1.4.1 operations

[80]:
```
d = {}
d['a'] = 1
d['b'] = 2
d
```

[80]:
```
{'a': 1, 'b': 2}
```

How many elements?

[81]:
```
len(d)
```

[81]:
```
2
```

what is the value of 'a'?

[82]:
```
d['a']
```

[82]:
```
1
```

let us remove `a`

[83]:
```
del d['a']              #
d
```

[83]:
```
{'b': 2}
```

membership is checked against the keys

[84]:
```
d['c'] = 3
'c' in d
```

[84]:
```
True
```

not the values

[85]:
```
3 in d                  # not the values
```

[85]:
```
False
```

obviously, we can also check the values...

[86]:
```
3 in d.values()
```

[86]:
```
True
```

have a look at dic properties

[87]:
```
dir(dict)
```

[87]:
```
['__class__',
 '__class_getitem__',
 '__contains__',
 '__delattr__',
 '__delitem__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__ior__',
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
 '__ne__',
 '__new__',
 '__or__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__reversed__',
```

```
 '__ror__',
 '__setattr__',
 '__setitem__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 'clear',
 'copy',
 'fromkeys',
 'get',
 'items',
 'keys',
 'pop',
 'popitem',
 'setdefault',
 'update',
 'values']
```

Clean everything

[88]:
```
d.clear()
d
```

[88]:
```
{}
```

1.4.2 dictionaries inline (optional)

[89]:
```
d = dict(zip('hello', range(5)))
d
```

[89]:
```
{'h': 0, 'e': 1, 'l': 3, 'o': 4}
```

[90]:
```
d.keys()
```

[90]:
```
dict_keys(['h', 'e', 'l', 'o'])
```

[91]:
```
d.values()
```

[91]:
```
dict_values([0, 1, 3, 4])
```

[92]:
```
d.items()
```

[92]:
```
dict_items([('h', 0), ('e', 1), ('l', 3), ('o', 4)])
```

1.4.3 getting items

removes a random item

[93]:
```
d.popitem()
```

[93]:
```
('o', 4)
```

remove item with key `l`

[94]:
```
d.pop('l')
```

[94]:
```
3
```

[95]:
```
d
```

[95]:
```
{'h': 0, 'e': 1}
```

remove a key not in dictionary -> KeyError

[96]:
```
d.pop('not-a-key')
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In [96], line 1
----> 1 d.pop('not-a-key')

KeyError: 'not-a-key'
```

if the key is not in the dictionary a default value can be returned

[97]:
```
d.pop('not-a-key', 'default-value')
```

[97]:
```
'default-value'
```

many way do we can update a dictionary

[98]:
```
d["Life of Brian"] = 1979
d.update({'And Now for Something Completely Different': 1971})
d.update(a = 1975)
d
```

[98]:
```
{'h': 0,
 'e': 1,
 'Life of Brian': 1979,
 'And Now for Something Completely Different': 1971,
 'a': 1975}
```

As seen, a way to get key's values is to use `d['a']`. Another way is

[99]:
```
d.get('a')
```

[99]:
```
1975
```

if the key is not in the dictionary a default value can be returned

[100]:
```
d.get('a', 177)
```

[100]: 1975

[101]:
```
d.get('b', 177)
```

[101]: 177

if key is inexistent, `None` is returned

[102]:
```
x = d.get('b')
print(x)
```

```
None
```

A value can be set if a key is not defined. The setdefault() method returns the value of the item with the specified key.

[103]:
```
d = {}
d.setdefault('a', 1)          # 'a' is missing, we get default value
```

[103]: 1

If the key does not exist, insert the key, with the specified value

[104]:
```
d.setdefault('a', 5)          # let's try to override the value
```

[104]: 1

[105]:
```
d
```

[105]: {'a': 1}

2 Exercises

Go here...

3 The collections module (optional)

When Python general purpose built-in containers (tuple, list, set, and dict) aren't enough, we can find specialized container data types in the collections module...

- `namedtuple` A factory function for creating tuple subclasses with named fields
- `deque` A list-like container with fast appends and pops on either end
- `ChainMap` A dict-like class for creating a single view of multiple mappings
- `Counter` A dict subclass for counting hashable objects
- `OrderedDict` A dict subclass that remembers the order entries were added
- `defaultdict` A dict subclass that calls a factory function to supply missing values
- `UserDict` A wrapper around dictionary objects for easier dict subclassing
- `UserList` A wrapper around list objects for easier list subclassing

- `UserString` A wrapper around string objects for easier string subclassing

See the manual for detailed descriptions and further containers.