Theory: Collections module

39 minutes 5 / 7 problems solved

The collections module provides data types similar to the built-in Python collections but with advanced features. You may have encountered some of them before, but in this topic, we will talk about `collections.OrderedDict`, `collections.namedtuple`, and `collections.ChainMap`.

§1. collections.OrderedDict

`OrderedDict` is a dictionary-like object that remembers the order of the given keys; it never changes them. Starting from version 3.7, a built-in Python dictionary preserves the order too, `OrderedDict` is now less important. However, there are still some useful features that make a big difference, we will discuss them in brief. `OrderedDict` was intended to be a better version of the dictionary in reordering operations and it lives up to it!

Let's say we want to create an `OrderedDict` with names of students as keys, and their average grades as values. The order in the dictionary is strict, from the most successful student to the least one, and is to be preserved as such. To do so, we can either use the `OrderedDict` constructor or get the `OrderedDict` from a regular dictionary:

```
from collections import OrderedDict

# this is the constructor
marks = OrderedDict()
marks['Smith'] = 9.5
marks['Brown'] = 8.1
marks['Moore'] = 7.4

print(marks)  # OrderedDict([('Smith', 9.5), ('Brown', 8.1), ('Moore', 7.4)])


# this is the conversion

my_dict = {'Smith': 9.5, 'Brown': 8.1, 'Moore': 7.4}

my_ord_dict = OrderedDict(my_dict)


print(my_ord_dict)  # OrderedDict([('Smith', 9.5), ('Brown', 8.1), ('Moore', 7.4)]
)
```

> The example with conversion is relevant only to Python 3.7 and higher where the information about the order is already remembered in regular dictionaries. In earlier Python versions, this procedure would make no sense as the order of elements in the dictionary we want to convert is arbitrary.

As `OrderedDict` resembles a regular dictionary a lot, we will only point out methods that set them apart.

1. While the `popitem()` method applied to a regular dictionary takes no argument, the same method for the `OrderedDict` can take an additional boolean parameter `last`. If `last=True`, the last key-value pair is returned and deleted from the `OrderedDict`, and if `last=False`, it is applied to the first pair.

Current topic:

- ✓ Collections module

Topic depends on:

- ✓ Dictionary (Stage 1) 6★
- ✓ Tuple

Table of contents:

```
print(my_ord_dict)  # OrderedDict([('Smith', 9.5), ('Brown', 8.1), ('Moore', 7.4)])

my_ord_dict.popitem(last=True)   # ('Moore', 7.4)
print(my_ord_dict)  # OrderedDict([('Smith', 9.5), ('Brown', 8.1)])

my_ord_dict.popitem(last=False)  # ('Smith', 9.5)
print(my_ord_dict)  # OrderedDict([('Brown', 8.1)])
```

2. The `move_to_end()` method takes arguments as a key and, again, the `last` parameter. If `last=True`, the key-value pair moves to the end of the `OrderedDict`, and if `last=False` — to the beginning.

```
print(my_ord_dict)  # OrderedDict([('Smith', 9.5), ('Brown', 8.1), ('Moore', 7.4)])

my_ord_dict.move_to_end('Brown', last=False)

print(my_ord_dict)  # OrderedDict([('Brown', 8.1), ('Smith', 9.5), ('Moore', 7.4)])

my_ord_dict.move_to_end('Smith', last=True)

print(my_ord_dict)  # OrderedDict([('Brown', 8.1), ('Moore', 7.4), ('Smith', 9.5)])
```

3. Finally, there is a difference in how the dictionaries are compared. With `OrderedDict`, two dictionaries are considered equal only if the order of their elements is the same; while with two built-in dictionaries, the order does not matter.

```
regular_dict_1 = {'Smith': 9.5, 'Brown': 8.1, 'Moore': 7.4}
regular_dict_2 = {'Brown': 8.1, 'Moore': 7.4, 'Smith': 9.5}
ordered_dict_1 = OrderedDict(regular_dict_1)
ordered_dict_2 = OrderedDict(regular_dict_2)

regular_dict_1 == regular_dict_2  # True
ordered_dict_1 == ordered_dict_2  # False
```

§2. collections.namedtuple

`collections.namedtuple` is a factory function to make subtypes of tuples with named elements. To create a named tuple, we first invoke the `namedtuple` function and then use the result as a template for our future items.

```
from collections import namedtuple

person_template = namedtuple('Person', ['name', 'age', 'occupation'])
```

In the example above, the subclass 'Person' is created. Its field names `'name'`, `'age'`, and `'occupation'` are in one list but they can also be defined in one string, separated by spaces or commas: `person_template = namedtuple('Person', 'name, age, occupation')`, or `person_template = namedtuple('Person', 'name age occupation')`.

Once we have the subclass, we can use the same template to create named tuple entities:

```
# field values can be defined either positionally or using the field names
mary = person_template('Mary', '25', 'doctor')
david = person_template(name='David', age='33', occupation='lawyer')

print(mary.name)   # Mary
print(david)       # Person(name='David', age='33', occupation='lawyer')
# the elements can also be accessed by their index, as in a regular tuple
print(david[2])    # lawyer
```

Note that the subclass name, 'Person' in our case, cannot be used itself to create instances; if we try to do so, a `NameError` will occur:

```
anna = Person(name='Anna', age='41', occupation='musician')

# Traceback (most recent call last):
#   File "<pyshell#5>", line 1, in <module>
#     anna = Person(name='Anna', age='41', occupation='musician')
# NameError: name 'Person' is not defined
```

Instead, we should always used the defined template, `person_template`.

A new named tuple can also be created from a list:

```
susanne = person_template._make(['Susanne', '23', 'journalist'])

print(susanne)  # Person(name='Susanne', age='23', occupation='journalist')
```

Named tuples allow us to replace field values with new ones, and we can see what fields are present in it. To do this, we should use the `_replace()` and `_fields()` methods:

```
mary = mary._replace(age='26')
print(mary)          # Person(name='Mary', age='26', occupation='doctor')
print(mary._fields)  # ('name', 'age', 'occupation')
```

> Note that the named tuple is immutable, just as regular tuples. This means that when we `_replace()` a field value, a new object is created instead of changing the existing one.

Finally, we can get an ordered dictionary out of named tuples with the help of the `_asdict()` function:

```
susanne_info = susanne._asdict()

print(susanne_info)  # OrderedDict([('name', 'Susanne'), ('age', '23'), ('occupation', 'journalist')])
```

§3. collections.ChainMap

Now, imagine you have created several dictionaries and want to analyze them and work with their data at once. Updating elements simultaneously in all dictionaries is not as easy as you would prefer, so the best decision is `ChainMap`. It allows you to make a collection of your dictionaries and, as a result, you will perform all operations on a collection instead of each separate dictionary.

```
from collections import ChainMap


laptop_labels = {'Lenovo': 600, 'Dell': 2000, 'Asus': 354}
operating_system = {'Windows': 2500, 'Linux': 400, 'MacOS': 54}
chain = ChainMap(laptop_labels, operating_system)

print(chain)  # ChainMap({'Lenovo': 600, 'Dell': 2000, 'Asus': 354}, {'Windows': 2500, 'Linux': 400, 'MacOS': 54})
```

You can access every item by key, as in the example below. You will get the value of the first key with the given name. If you change a value in one dictionary, this information in the chain will be changed too.

```
operating_system['Linux'] = 450  # changing a value in a dictionary
print(chain['Linux'])            # 450
```

Speaking of methods, we can use usual dictionary methods and some peculiar ones to work with the chained objects. For example, there is the `new_child()` method that allows you to add another dictionary in your chain, and you will get a new structure with another dictionary added:

```
processor = {'Celeron': 600, 'Pentium': 2000, 'Ryzen 5': 354}
new_chain = chain.new_child(processor)

print(new_chain)  # ChainMap({'Celeron': 600, 'Pentium': 2000, 'Ryzen 5': 354}, {'Lenovo': 600, 'Dell': 2000, 'Asus': 354}, {'Windows': 2500, 'Linux': 400, 'MacOS': 54})
```

The `maps` method allows you to get access to a certain dictionary by its index:

```

print(new_chain.maps[1])  # ChainMap({'Lenovo': 600, 'Dell': 2000, 'Asus': 354})
```

The method `parents` gets rid of the first dictionary and returns the rest:

```

print(new_chain)      # ChainMap({'Celeron': 600, 'Pentium': 2000, 'Ryzen 5': 354}, {'Lenovo': 600, 'Dell': 2000, 'Asus': 354}, {'Windows': 2500, 'Linux': 400, 'MacOS': 54})
without_first = new_chain.parents

print(without_first)  # ChainMap({'Lenovo': 600, 'Dell': 2000, 'Asus': 354}, {'Windows': 2500, 'Linux': 400, 'MacOS': 54})
```

Note that the methods `new_child` and `parents` do not change the chain itself, they return a new object, so if we want to work with them further, we need to assign them to a variable.

§4. Conclusion

In this topic, we've seen how `OrderedDict` is different from the regular dictionary; how you can work with `namedtuple` to store different data in one place, and how `ChainMap` can be used.

To sum up,

- `collections.OrderedDict`'s `popitem` method differs from the one in the dictionary by having a parameter `last`, and there is one more method `move_to_end` which changes the position of the item given to it;
- `collections.namedtuple` keeps diverse information about an object, and any field containing this information can be accessed by name;
- `ChainMap` serves as a container for several independent dictionaries, and allows you to work with them at once.

For additional information about `OrderedDict`, `namedtuple`, and `ChainMap` you can check out the official Python documentation. We hope that you'll use these objects and methods in your projects!

Report a typo

72 users liked this theory. **5** didn't like it. **What about you?**

Start practicing

This content was created 7 months ago and updated about 18 hours ago. Share your feedback below in comments to help us improve it!

Comments (6) Hints (0) Useful links (0) Show discussion