Python → Collections → Defaultdict and Counter

Theory: Defaultdict and Counter

43 minutes 5 / 7 problems solved

Start practicing

689 users solved this topic. Latest completion was about 12 hours ago.

Current topic:

✓ Defaultdict and Counter ···

Topic depends on:

✓ Dictionary Stage 1 6★ ···

✓ Split and join Stage 1 8★ ···

✓ Load module Stage 1 5★ ···

Table of contents:

↑ Defaultdict and Counter

§1. Method dict.setdefault()

§2. collections.defaultdict

§3. collections.Counter

§4. Summary

Feedback & Comments

Dictionary is an incredibly versatile data structure. However, it has its limitations. In this topic, we'll look at one special `dict` method that makes working with dictionaries much easier as well as some dictionary-like objects useful in particular situations.

§1. Method dict.setdefault()

Often dictionaries are used to store frequencies of elements from some collection. You may want to store word counts from a text, or values from a list. How would you do it? Well, the obvious way is to loop through all elements and increment the count of the element you've encountered by one. However, you have to check if the element has already been encountered otherwise you may get a `KeyError`. This may be tedious: you have to write an `if` statement, or catch exceptions, or handle `None` values (if you use `dict.get()`).

As you might have guessed, there are easier ways to deal with this. One of them is the `dict.setdefault()` method.

Let's check out an example. Suppose, we want to create a frequency dictionary for the text below.

```
# long string with text
text = ("Gambit is a chess opening in which a player risks one or more pawns "
        "or a minor piece to gain an advantage in position")

# convert text to list of words
text_list = text.lower().split()
```

How can we use `dict.setdefault()` to create a frequency dictionary? Well, the method takes two parameters: `key` and optional `default`. It returns the value of the key and if it is not present, sets it to default (or `None`). Following this logic, for the frequency dictionary, the default value should be 0.

```
# create empty dictionary
freq_dict = {}

# loop through text and count words
for word in text_list:
    # set the default value to 0
    freq_dict.setdefault(word, 0)
    # increment the value by 1
    freq_dict[word] += 1



# examples

print(freq_dict["gambit"])  # 1

print(freq_dict["a"])  # 3
```

As you can see, we've successfully created a frequency dictionary in virtually 3 lines of code! No need to handle exceptions or check conditions.

The `setdefault()` method is particularly useful when the value is a mutable object, for instance, a list or even another dictionary. Suppose, we want to create a dictionary of indexes of all occurrences in the list. This is how it can be done.

```
index_dict = {}

for index, word in enumerate(text_list):
    index_dict.setdefault(word, []).append(index)

# example
print(index_dict["or"])  # [11, 14]
```

Here we use `enumerate` function to iterate over elements of `text_list` and get their indexes at the same time. Since `dict.setdefault()` returns the value of the key we can directly append the new value to the list.

This method is also handy when dealing with missing values. If we're not sure that some object is a key in the dictionary, we can just set its value in the process:

```
print(freq_dict.setdefault("are", 0))  # 0
```

As a result of the code above, the word "are" will be added to `freq_dict` with the value 0.

§2. collections.defaultdict

Along with the `dict.setdefault()` method there is another way to manage default values: `defaultdict` from the `collections` module.

`defaultdict` is a dictionary-like object defined in `collections`. The only significant difference is that when creating the new dictionary we can specify a `default_factory` parameter. `default_factory` should be a function-like object that determines the type or the value of the parameter. For example, the `default_factory` can be a list function: `list`. In this case, the default value of any object will be an empty list.

In our example with the frequency dictionary, the `default_factory` should be `int`: the default value of the `int()` function is 0. Let's see how that can be used:

```
from collections import defaultdict

freq_defaultdict = defaultdict(int)

for word in text_list:
    freq_defaultdict[word] += 1

print(freq_defaultdict["chess"])  # 1
```

Note that you shouldn't call the function object you're setting as `default_factory`!

`defaultdict` also makes it extremely easy to deal with missing keys because we have already defined a default value for any object:

```
print(freq_defaultdict["python"])  # 0
```

As you can see, we don't really need to do anything: if you've defined the `default_factory` parameter, dealing with missing keys doesn't cause any errors. Like with `dict.setdefault()`, the missing keys are added to the `defaultdict` if we try to check their values.

§3. collections.Counter

Now, all the ways of creating frequency dictionaries we've covered so far work perfectly fine. However, they require a bit of work, that is creating a dictionary and going through the list and counting the elements. You'll be happy to know that there is a very straightforward way to do it without all the work: `Counter` object from `collections`.

`Counter` is another dictionary-like object used specifically for counting the elements of an iterable object. Take our example again:

```
from collections import Counter

freq_counter = Counter(text_list)

print(freq_counter)

# Counter({'a': 3, 'in': 2, 'or': 2, 'gambit': 1, 'is': 1, 'chess': 1, 'opening': 1,
# 'which': 1, 'player': 1, 'risks': 1, 'one': 1, 'more': 1, 'pawns': 1, 'minor': 1,
# 'piece': 1, 'to': 1, 'gain': 1, 'an': 1, 'advantage': 1, 'position': 1})
```

That's all it takes to create a `Counter` which is essentially a frequency dictionary!

Another advantage of `Counter` is that the count of the missing object is 0, no `KeyErrors` are raised! However, the missing object isn't added to the dictionary if we try to check its value.

```
print(freq_counter["alpaca"])  # 0
```

`Counter` supports most of `dict` methods and it also has some special methods of its own. For example, the `most_common(n)` method which returns a list of tuples of n most common elements with their counts. If n isn't specified, all elements are returned in the descending order of their frequencies.

```
print(freq_counter.most_common(5))
# [('a', 3), ('in', 2), ('or', 2), ('gambit', 1), ('is', 1)]
print(freq_counter.most_common())
# [('a', 3), ('in', 2), ('or', 2), ('gambit', 1), ('is', 1), ('chess', 1), ('opening', 1),
# ('which', 1), ('player', 1), ('risks', 1), ('one', 1), ('more', 1), ('pawns', 1),
# ('minor', 1), ('piece', 1), ('to', 1), ('gain', 1), ('an', 1), ('advantage', 1), ('position', 1)]
```

Note that since Python 3.7, `Counter` preserves the insertion order of elements (just like `dict`). So, elements with the same count are returned in the order of their appearance in the original collection.

§4. Summary

In this topic, we've seen how you can define default values for dictionary keys, deal with missing keys and create different types of frequency dictionaries.

To sum up,

- `dict.setdefault` is used to set a default value for the particular key;
- `collections.defaultdict` defines a default value for all objects;
- `collections.Counter` counts elements of an iterable object.

For additional information about `defaultdict` and `Counter` you can check out the official Python documentation.

We hope that you'll use these objects and methods in your projects!

Report a typo

83 users liked this theory. 0 didn't like it. **What about you?**

😍 🙂 😐 🙁 😡

Start practicing

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