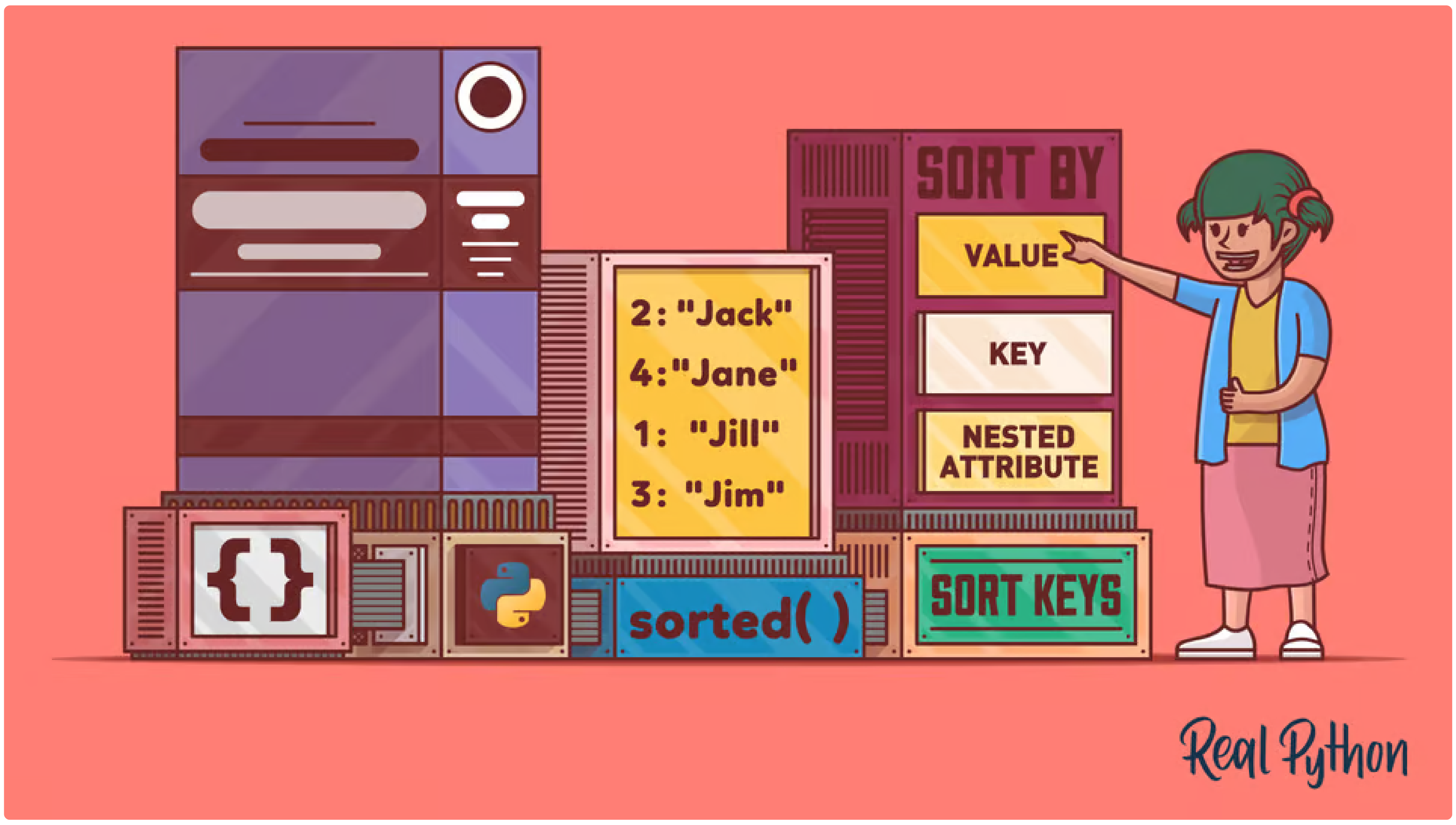

Sorting a Python Dictionary: Values, Keys, and More

by Ian Currie Aug 10, 2022 1 Comment data-structures intermediate

Mark as Completed

Share Share Email

Table of Contents

You've got a dictionary, but you'd like to sort the key-value pairs. Perhaps you've tried passing a dictionary to the `sorted()` function but haven't gotten the results you expected. In this tutorial, you'll go over everything you need to know if you want to sort dictionaries in Python.

In this tutorial, you'll:

- Review how to use the **`sorted()`** function
- Learn how to get dictionary **views** to **iterate** over
- Understand how dictionaries are cast to **lists** during sorting
- Learn how to specify a **sort key** to sort a dictionary by value, key, or nested attribute
- Review dictionary **comprehensions** and the `dict()` **constructor** to rebuild your dictionaries
- Consider alternative **data structures** for your **key-value data**

Along the way, you'll also use the `timeit` module to time your code and get tangible results for comparing the different methods of sorting key-value data. You'll also consider whether a sorted dictionary is really your best option, as it's not a particularly common pattern.

To get the most out of this tutorial, you should know about dictionaries, lists, tuples, and functions. With that knowledge, you'll be able to sort dictionaries by the end of this tutorial. Some exposure to higher-order functions, such as lambda functions, will also come in handy but isn't a requirement.

Free Download: Click here to download the code that you'll use to sort key-value pairs in this tutorial.

First up, you'll learn some foundational knowledge before trying to sort a dictionary in Python.

Rediscovering Dictionary Order in Python

Before Python 3.6, dictionaries were inherently **unordered**. A Python dictionary is an implementation of the hash table, which is traditionally an unordered data structure.

As a side effect of the compact dictionary implementation in Python 3.6, dictionaries started to conserve insertion order. From 3.7, that insertion order has been *guaranteed*.

If you wanted to keep an ordered dictionary as a data structure before compact dictionaries, then you could use `OrderedDict` from the `collections` module. Similar to the modern compact dictionary, it also keeps insertion order, but neither type of dictionary sorts itself.

Another alternative for storing an ordered key-value pair data is to store the pairs as a list of tuples. As you'll see later in the tutorial, using a list of tuples could be the best choice for your data.

An essential point to understand when sorting dictionaries is that even though they conserve insertion order, they're not considered a sequence. A dictionary is like a set of key-value pairs, and sets are unordered.

Dictionaries also don't have much reordering functionality. They're not like lists, where you can **insert** elements at any position. In the next section, you'll explore the consequences of this limitation further.

Understanding What Sorting A Dictionary Really Means

Because dictionaries don't have much reordering functionality, when sorting a dictionary, it's rarely done **in-place**. In fact, there are no methods for explicitly moving items in a dictionary.

If you wanted to sort a dictionary in-place, then you'd have to use the `del` keyword to delete an item from the dictionary and then add it again. Deleting and then adding again effectively moves the key-value pair to the end.

The `OrderedDict` class has a specific method to move an item to the end or the start, which may make `OrderedDict` preferable for keeping a sorted dictionary. However, it's still not very common and isn't very performant, to say the least.

The typical method for sorting dictionaries is to get a dictionary **view**, sort it, and then cast the resulting list back into a dictionary. So you effectively go from a dictionary to a list and back into a dictionary. Depending on your use case, you may not need to convert the list back into a dictionary.

Note: Sorted dictionaries aren't a very common pattern. You'll explore more about that topic later in the tutorial.

With those preliminaries out of the way, you'll get to sorting dictionaries in the next section.

Sorting Dictionaries in Python

In this section, you'll be putting together the components of sorting a dictionary so that, in the end, you can master the most common way of sorting a dictionary:

Python

```
>>> people = {3: "Jim", 2: "Jack", 4: "Jane", 1: "Jill"}

>>> # Sort by key
>>> dict(sorted(people.items()))
{1: 'Jill', 2: 'Jack', 3: 'Jim', 4: 'Jane'}

>>> # Sort by value
>>> dict(sorted(people.items(), key=lambda item: item[1]))
{2: 'Jack', 4: 'Jane', 1: 'Jill', 3: 'Jim'}
```

Don't worry if you don't understand the snippets above—you'll review it all step-by-step in the following sections. Along the way, you'll learn how to use the `sorted()` function with sort keys, `lambda` functions, and dictionary constructors.

Using the `sorted()` Function

The critical function that you'll use to sort dictionaries is the built-in `sorted()` function. This function takes an iterable as the main argument, with two optional keyword-only arguments—a `key` function and a `reverse` Boolean value.

To illustrate the `sorted()` function's behavior in isolation, examine its use on a list of numbers:

Python

```
>>> numbers = [5, 3, 4, 3, 6, 7, 3, 2, 3, 4, 1]
>>> sorted(numbers)
[1, 2, 3, 3, 3, 3, 4, 4, 5, 6, 7]
```

As you can see, the `sorted()` function takes an iterable, sorts **comparable** elements like numbers in **ascending** order, and returns a new list. With strings, it sorts them in **alphabetical** order:

Python

```
>>> words = ["aa", "ab", "ac", "ba", "cb", "ca"]
>>> sorted(words)
['aa', 'ab', 'ac', 'ba', 'ca', 'cb']
```

Sorting by numerical or alphabetical precedence is the most common way to sort elements, but maybe you need more control.

Say you want to sort on the *second character* of each word in the last example. To customize what the `sorted()` function uses to sort the elements, you can pass in a callback function to the `key` parameter.

A callback function is a function that's passed as an argument to another function. For `sorted()`, you pass it a function that acts as a sort key. The `sorted()` function will then *call back* the sort key for every element.

In the following example, the function passed as the key accepts a string and will return the second character of that string:

Python

```
>>> def select_second_character(word):
...     return word[1]
...
>>> sorted(words, key=select_second_character)
['aa', 'ba', 'ca', 'ab', 'cb', 'ac']
```

The `sorted()` function passes every element of the `words` iterable to the `key` function and uses the return value for comparison. Using the key means that the `sorted()` function will compare the second letter instead of comparing the whole string directly.

More examples and explanations of the `key` parameter will come later in the tutorial when you use it to sort dictionaries by values or nested elements.

If you take another look at the results of this last sorting, you may notice the stability of the `sorted()` function. The three elements, `aa`, `ba` and `ca`, are equivalent when sorted by their second character. Because they're equal, the `sorted()` function conserves their *original order*. Python guarantees this stability.

Note: Every list also has a `.sort()` method, which has the same signature as the `sorted()` function. The main difference is that the `.sort()` method sorts the list **in-place**. In contrast, the `sorted()` function returns a new list, leaving the original list unmodified.

You can also pass `reverse=True` to the sorting function or method to return the reverse order. Alternatively, you can use the `reversed()` function to invert the iterable after sorting:

Python

```
>>> list(reversed([3, 2, 1]))
[1, 2, 3]
```

If you want to dive deeper into the mechanics of sorting in Python and learn how to sort data types other than dictionaries, then check out the tutorial on how to use `sorted()` and `.sort()`.

So, how about dictionaries? You can actually take the dictionary and feed it straight into the `sorted()` function:

Python

```
>>> people = {3: "Jim", 2: "Jack", 4: "Jane", 1: "Jill"}
>>> sorted(people)
[1, 2, 3, 4]
```

But the default behavior of passing in a dictionary directly to the `sorted()` function is to take the **keys** of the dictionary, sort them, and return a **list** of the keys *only*. That's probably not the behavior you had in mind! To preserve all the information in a dictionary, you'll need to be acquainted with **dictionary views**.

Getting Keys, Values, or Both From a Dictionary

If you want to conserve all the information from a dictionary when sorting it, the typical first step is to call the `.items()` method on the dictionary. Calling `.items()` on the dictionary will provide an iterable of tuples representing the key-value pairs:

Python

```
>>> people = {3: "Jim", 2: "Jack", 4: "Jane", 1: "Jill"}
>>> people.items()
dict_items([(3, 'Jim'), (2, 'Jack'), (4, 'Jane'), (1, 'Jill')])
```

The `.items()` method returns a read-only dictionary view object, which serves as a window into the dictionary. This view is *not* a copy or a list—it's a read-only iterable that's actually *linked* to the dictionary it was generated from:

Python

```
>>> people = {3: "Jim", 2: "Jack", 4: "Jane", 1: "Jill"}
>>> view = people.items()
>>> people[2] = "Elvis"
>>> view
dict_items([(3, 'Jim'), (2, 'Elvis'), (4, 'Jane'), (1, 'Jill')])
```

You'll notice that any updates to the dictionary also get reflected in the view because they're linked. A view represents a lightweight way to iterate over a dictionary without generating a list first.

Note: You can use `.values()` to get a view of the values only and `.keys()` to get one with only the keys.

Crucially, you can use the `sorted()` function with dictionary views. You call the `.items()` method and use the result as an argument to the `sorted()` function. Using `.items()` keeps all the information from the dictionary:

Python

```
>>> people = {3: "Jim", 2: "Jack", 4: "Jane", 1: "Jill"}
>>> sorted(people.items())
[(1, 'Jill'), (2, 'Jack'), (3, 'Jim'), (4, 'Jane')]
```

This example results in a sorted list of tuples, with each tuple representing a key-value pair of the dictionary.

If you want to end up with a dictionary sorted by values, then you've still got two issues. The default behavior still seems to sort by *key* and not value. The other issue is that you end up with a *list* of tuples, not a dictionary. First, you'll figure out how to sort by value.

Understanding How Python Sorts Tuples

When using the `.items()` method on a dictionary and feeding it into the `sorted()` function, you're passing in an iterable of tuples, and the `sorted()` function compares the entire tuple directly.

When comparing tuples, Python behaves a lot like it's sorting strings alphabetically. That is, it sorts them lexicographically.

Lexicographical sorting means that if you have two tuples, `(1, 2, 4)` and `(1, 2, 3)`, then you start by comparing the first item of each tuple. The first item is `1` in both cases, which is equal. The second element, `2`, is also identical in both cases. The third elements are `4` and `3`, respectively. Since `3` is less than `4`, you've found which item is *less* than the other.

So, to order the tuples `(1, 2, 4)` and `(1, 2, 3)` lexicographically, you would switch their order to `(1, 2, 3)` and `(1, 2, 4)`.

Because of Python's lexicographic sorting behavior for tuples, using the `.items()` method with the `sorted()` function will always sort by keys unless you use something extra.

Using the `key` Parameter and Lambda Functions

For example, if you want to sort by value, then you have to specify a **sort key**. A sort key is a way to extract a comparable value. For instance, if you have a pile of books, then you might use the author surname as the sort key. With the `sorted()` function, you can specify a sort key by passing a callback function as a `key` argument.

Note: The `key` argument has nothing to do with a dictionary key!

To see a sort key in action, take a look at this example, which is similar to the one you saw in the section introducing the `sorted()` function:

Python

```
>>> people = {3: "Jim", 2: "Jack", 4: "Jane", 1: "Jill"}

>>> # Sort key
>>> def value_getter(item):
...     return item[1]
...

>>> sorted(people.items(), key=value_getter)
[(2, 'Jack'), (4, 'Jane'), (1, 'Jill'), (3, 'Jim')]

>>> # Or with a lambda function
>>> sorted(people.items(), key=lambda item: item[1])
[(2, 'Jack'), (4, 'Jane'), (1, 'Jill'), (3, 'Jim')]
```

In this example, you try out two ways of passing a `key` parameter. The `key` parameter accepts a callback function. The function can be a normal function identifier or a lambda function. The lambda function in the example is the exact equivalent of the `value_getter()` function.

Note: Lambda functions are also known as anonymous functions because they don't have a name. Lambda functions are standard for functions that you're only using once in your code.

Lambda functions confer no benefit apart from making things more compact, eliminating the need to define a function separately. They keep things nicely contained on the same line:

Python

```python
# With a normal function
def value_getter(item):
    return item[1]

sorted(people.items(), key=value_getter)

# With a lambda function
sorted(people.items(), key=lambda item: item[1])
```

For basic getter functions like the one in the example, lambdas can come in handy. But lambdas can make your code less readable for anything more complex, so use them with care.

Lambdas can also only ever contain exactly one expression, making any multiline statements like `if` statements or `for` loops off limits. You can work around this by using comprehensions and `if` expressions, for example, but those can make for long and cryptic one-liners.

The `key` callback function will receive each element of the iterable that it's sorting. The callback function's job is to *return something that can be compared*, such as a number or a string. In this example, you named the function `value_getter()` because all it does is get the value from a key-value tuple.

Since the default behavior of `sorted()` with tuples is to sort lexicographically, the `key` parameter allows you to select a value from the element that it's comparing.

In the next section, you'll take sort keys a bit further and use them to sort by a nested value.

Selecting a Nested Value With a Sort Key

You can also go further and use a sort key to select nested values that may or may not be present and return a default value if they're not present:

Python

```python
data = {
    193: {"name": "John", "age": 30, "skills": {"python": 8, "js": 7}},
    209: {"name": "Bill", "age": 15, "skills": {"python": 6}},
    746: {"name": "Jane", "age": 58, "skills": {"js": 2, "python": 5}},
    109: {"name": "Jill", "age": 83, "skills": {"java": 10}},
    984: {"name": "Jack", "age": 28, "skills": {"c": 8, "assembly": 7}},
    765: {"name": "Penelope", "age": 76, "skills": {"python": 8, "go": 5}},
    598: {"name": "Sylvia", "age": 62, "skills": {"bash": 8, "java": 7}},
    483: {"name": "Anna", "age": 24, "skills": {"js": 10}},
    277: {"name": "Beatriz", "age": 26, "skills": {"python": 2, "js": 4}},
}

def get_relevant_skills(item):
    """Get the sum of Python and JavaScript skill"""
    skills = item[1]["skills"]

    # Return default value that is equivalent to no skill
    return skills.get("python", 0) + skills.get("js", 0)

print(sorted(data.items(), key=get_relevant_skills, reverse=True))
```

In this example, you have a dictionary with numeric keys and a nested dictionary as a value. You want to sort by the combined Python and JavaScript skills, attributes found in the `skills` subdictionary.

Part of what makes sorting by the combined skill tricky is that the `python` and `js` keys aren't present in the `skills` dictionary for all people. The `skills` dictionary is also nested. You use `.get()` to read the keys and provide `0` as a default value that's used for missing skills.

You've also used the `reverse` argument because you want the top Python skills to appear first.

Note: You didn't use a lambda function in this example. While it's possible, it would make for a long line of potentially cryptic code:

Python

```python
sorted(
    data.items(),
    key=lambda item: (
        item[1]["skills"].get("python", 0)
        + item[1]["skills"].get("js", 0)
    ),
    reverse=True,
)
```

A lambda function can only contain one expression, so you repeat the full look-up in the nested `skills` subdictionary. This inflates the line length considerably.

The lambda function also requires multiple chained square bracket (`[]`) indices, making it harder to read than necessary. Using a lambda in this example only saves a few lines of code, and the performance difference is negligible. So, in these cases, it usually makes more sense to use a normal function.

You've successfully used a higher-order function as a sort key to sort a dictionary view by value. That was the hard part. Now there's only one issue left to solve—converting the list that `sorted()` yields back into a dictionary.

Converting Back to a Dictionary

The only issue left to address with the default behavior of `sorted()` is that it returns a list, not a dictionary. There are a few ways to convert a list of tuples back into a dictionary.

You can iterate over the result with a `for` loop and populate a dictionary on each iteration:

Python

```
>>> people = {3: "Jim", 2: "Jack", 4: "Jane", 1: "Jill"}
>>> sorted_people = sorted(people.items(), key=lambda item: item[1])

>>> sorted_people_dict = {}
>>> for key, value in sorted_people:
...     sorted_people_dict[key] = value
...

>>> sorted_people_dict
{2: 'Jack', 4: 'Jane', 1: 'Jill', 3: 'Jim'}
```

This method gives you absolute control and flexibility in deciding how you want to construct your dictionary. This method can be quite lengthy to type out, though. If you don't have any special requirements for constructing your dictionary, then you may want to go for a dictionary constructor instead:

Python

```
>>> people = {3: "Jim", 2: "Jack", 4: "Jane", 1: "Jill"}
>>> sorted_people = sorted(people.items(), key=lambda item: item[1])
>>> dict(sorted_people)
{2: 'Jack', 4: 'Jane', 1: 'Jill', 3: 'Jim'}
```

That's nice and compact! You could also use a dictionary comprehension, but that only makes sense if you want to change the shape of the dictionary or swap the keys and values, for example. In the following comprehension, you swap the keys and values:

Python

```
>>> {
...     value: key
...     for key, value in sorted(people.items(), key=lambda item: item[1])
... }
...
{'Jack': 2, 'Jane': 4, 'Jill': 1, 'Jim': 3}
```

Depending on how familiar you or your team are with comprehensions, this may be less readable than just using a normal `for` loop.

Congratulations, you've got your sorted dictionary! You can now sort it by any criteria that you'd like.

Now that you can sort your dictionary, you might be interested in knowing if there are any performance implications to using a sorted dictionary, or whether there are alternative data structures for key-value data.

Considering Strategic and Performance Issues

In this section, you'll be taking a quick peek at some performance tweaks, strategic considerations, and questions to ask yourself about how you'll use your key-value data.

Note: If you decide to go for an ordered collection, check out the Sorted Containers package, which includes a `SortedDict`.

You'll be leveraging the `timeit` module to get some metrics to work with. It's important to bear in mind that to make any solid conclusions about performance, you need to test on a variety of hardware, and with a variety of sample types and sizes.

Finally, note that you won't be going into detail about how to use `timeit`. For that, check out the tutorial on Python timers. You'll have some examples to play with, though.

Using Special Getter Functions to Increase Performance and Readability

You may have noticed that most of the sort key functions that you've used so far aren't doing very much. All the function does is get a value from a tuple. Making a getter function is such a common pattern that Python has a special way to create special functions that get values more quickly than regular functions.

The `itemgetter()` function can produce highly efficient versions of getter functions.

You pass `itemgetter()` an argument, which is typically the key or index position that you want to select. The `itemgetter()` function will then return a getter object that you call like a function.

That's right, it's a function that returns a function. Using the `itemgetter()` function is another example of working with higher-order functions.

The getter object from `itemgetter()` will call the `.__getitem__()` method on the item that's passed to it. When something makes a call to `.__getitem__()`, it needs to pass in the key or index of what to get. The argument that's used for `.__getitem__()` is the same argument that you passed to `itemgetter()`:

Python

```
>>> item = ("name", "Guido")

>>> from operator import itemgetter

>>> getter = itemgetter(0)
>>> getter(item)
'name'
>>> getter = itemgetter(1)
>>> getter(item)
'Guido'
```

In the example, you start off with a tuple, similar to one that you might get as part of a dictionary view.

You make the first getter by passing `0` as an argument to `itemgetter()`. When the resultant getter receives the tuple, it returns the first item in the tuple—the value at index `0`. If you call `itemgetter()` with an argument of `1`, then it gets the value at index position `1`.

You can use this itemgetter as a key for the `sorted()` function:

Python

```
>>> from operator import itemgetter

>>> fruit_inventory = [
...     ("banana", 5), ("orange", 15), ("apple", 3), ("kiwi", 0)
... ]

>>> # Sort by key
>>> sorted(fruit_inventory, key=itemgetter(0))
[('apple', 3), ('banana', 5), ('kiwi', 0), ('orange', 15)]

>>> # Sort by value
>>> sorted(fruit_inventory, key=itemgetter(1))
[('kiwi', 0), ('apple', 3), ('banana', 5), ('orange', 15)]

>>> sorted(fruit_inventory, key=itemgetter(2))
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    sorted(fruit_inventory, key=itemgetter(2))
IndexError: tuple index out of range
```

In this example, you start by using `itemgetter()` with `0` as an argument. Since it's operating on each tuple from the `fruit_inventory` variable, it gets the first element from each tuple. Then the example demonstrates initializing an `itemgetter` with `1` as an argument, which selects the second item in the tuple.

Finally, the example shows what would happen if you used `itemgetter()` with `2` as an argument. Since these tuples only have two index positions, trying to get the third element, with index `2`, results in a `IndexError`.

You can use the function produced by `itemgetter()` in place of the getter functions that you've been using up until now:

Python

```
>>> people = {3: "Jim", 2: "Jack", 4: "Jane", 1: "Jill"}

>>> from operator import itemgetter
>>> sorted(people.items(), key=itemgetter(1))
[(2, 'Jack'), (4, 'Jane'), (1, 'Jill'), (3, 'Jim')]
```

The `itemgetter()` function produces a function that has exactly the same effect as the `value_getter()` function from previous sections. The main reason you'd want to use the function from `itemgetter()` is because it's more efficient. In the next section, you'll start to put some numbers on just how much more efficient it is.

Measuring Performance When Using `itemgetter()`

So, you end up with a function that behaves like the original `value_getter()` from the previous sections, except that the version returned from `itemgetter()` is more efficient. You can use the `timeit` module to compare their performance:

Python

```python
# compare_lambda_vs_getter.py

from timeit import timeit

dict_to_order = {
    1: "requests",
    2: "pip",
    3: "jinja",
    4: "setuptools",
    5: "pandas",
    6: "numpy",
    7: "black",
    8: "pillow",
    9: "pyparsing",
    10: "boto3",
    11: "botocore",
    12: "urllib3",
    13: "s3transfer",
    14: "six",
    15: "python-dateutil",
    16: "pyyaml",
    17: "idna",
    18: "certifi",
    19: "typing-extensions",
    20: "charset-normalizer",
    21: "awscli",
    22: "wheel",
    23: "rsa",
}

sorted_with_lambda = "sorted(dict_to_order.items(), key=lambda item: item[1])"
sorted_with_itemgetter = "sorted(dict_to_order.items(), key=itemgetter(1))"

sorted_with_lambda_time = timeit(stmt=sorted_with_lambda, globals=globals())
sorted_with_itemgetter_time = timeit(
    stmt=sorted_with_itemgetter,
    setup="from operator import itemgetter",
    globals=globals(),
)

print(
    f"""\
{sorted_with_lambda_time=:.2f} seconds
{sorted_with_itemgetter_time=:.2f} seconds
itemgetter is {(
    sorted_with_lambda_time / sorted_with_itemgetter_time
):.2f} times faster"""
)
```

This code uses the `timeit` module to compare the sorting processes of the function from `itemgetter()` and a lambda function.

Running this script from the shell should give you similar results to what's below:

Shell

```shell
$ python compare_lambda_vs_getter.py
sorted_with_lambda_time=1.81 seconds
sorted_with_itemgetter_time=1.29 seconds
itemgetter is 1.41 times faster
```

A savings of around 40 percent is significant!

Bear in mind that when timing code execution, times can vary significantly between systems. That said, in this case, the ratio should be relatively stable across systems.

From the results of this test, you can see that using `itemgetter()` is preferable from a performance standpoint. Plus, it's part of the Python standard library, so there's no cost to using it.

Note: The difference between using a lambda and a normal function as the sort key is negligible in this test.

Do you want to compare the performance of some operations that you haven't covered here? Be sure to share the results by posting them in the comments!

Now you can squeeze a bit more performance out of your dictionary sorting, but it's worth taking a step back and considering whether using a sorted dictionary as your preferred data structure is the best choice. A sorted dictionary isn't a very common pattern, after all.

Coming up, you'll be asking yourself some questions about what you what you want to do with your sorted dictionary and whether it's the best data structure for your use case.

Judging Whether You Want to Use a Sorted Dictionary

If you're considering making a sorted key-value data structure, then there are a few things you might want to take into consideration.

If you're going to be adding data to a dictionary, and you want it to stay sorted, then you might be better off using a structure like a list of tuples or a list of dictionaries:

Python

```python
# Dictionary
people = {3: "Jim", 2: "Jack", 4: "Jane", 1: "Jill"}

# List of tuples
people = [
    (3, "Jim"),
    (2, "Jack"),
    (4, "Jane"),
    (1, "Jill")
]

# List of dictionaries
people = [
    {"id": 3, "name": "Jim"},
    {"id": 2, "name": "Jack"},
    {"id": 4, "name": "Jane"},
    {"id": 1, "name": "Jill"},
]
```

A list of dictionaries is the most widespread pattern because of its cross-language compatibility, known as language interoperability.

Language interoperability is especially relevant if you create an HTTP REST API, for instance. Making your data available over the Internet will likely mean serializing it in JSON.

If someone using JavaScript were to consume JSON data from a REST API, then the equivalent data structure would be an object. The kicker is that JavaScript objects are *not ordered*, so the order would end up scrambled!

This scrambling behavior would be true for many languages, and objects are even defined in the JSON specification as an unordered data structure. So, if you took care to order your dictionary before serializing to JSON, it wouldn't matter by the time it got into most other environments.

Note: Signposting an ordered sequence of key-value pairs may not only be relevant for serializing Python dictionaries into JSON. Imagine you have people on your team who are used to other languages. An ordered dictionary might be a foreign concept to them, so you may need to be explicit about the fact that you've created an ordered data structure.

One way to be explicit about having an ordered dictionary in Python is to use the aptly named `OrderedDict`.

Another option is to simply not worry about ordering the data if you don't need to. Including `id`, `priority`, or other equivalent attributes for each object can be enough to express order. If the ordering gets mixed up for any reason, then there'll always be an unambiguous way to sort it:

Python

```python
people = {
    3: {"priority": 2, "name": "Jim"},
    2: {"priority": 4, "name": "Jack"},
    4: {"priority": 1, "name": "Jane"},
    1: {"priority": 2, "name": "Jill"}
}
```

With a `priority` attribute, for instance, it's clear that `Jane` should be first in line. Being clear about your intended ordering is nicely in agreement with the old Python adage of *explicit is better than implicit*, from the Zen of Python.

What are the performance trade-offs with using a list of dictionaries versus a dictionary of dictionaries, though? In the next section, you'll start to get some data on just that very question.

Comparing the Performance of Different Data Structures

If performance is a consideration—maybe you'll be working with large datasets, for example—then you should carefully consider what you'll be doing with the dictionary.

The two main questions you'll seek to answer in the next few sections are:

1. Will you be sorting once and then making lots of lookups?
2. Will you be sorting many times and making very few lookups?

Once you've decided what usage patterns you'll be subjecting your data structure to, then you can use the `timeit` module to test the performance. These measurements can vary a lot with the exact shape and size of the data being tested.

In this example, you'll be pitting a dictionary of dictionaries against a list of dictionaries to see how they differ in terms of performance. You'll be timing sorting operations and lookup operations with the following sample data:

Sample Data Show/Hide

Each data structure has the same information, except one is structured as a dictionary of dictionaries, and the other is a list of dictionaries. First up, you'll be getting some metrics on the performance of sorting these two data structures.

Comparing the Performance of Sorting

In the following code, you'll be using `timeit` to compare the time it takes to sort the two data structures by the `age` attribute:

Python

```python
# compare_sorting_dict_vs_list.py

from timeit import timeit
from samples import dictionary_of_dictionaries, list_of_dictionaries

sorting_list = "sorted(list_of_dictionaries, key=lambda item:item['age'])"
sorting_dict = """
dict(
    sorted(
        dictionary_of_dictionaries.items(), key=lambda item: item[1]['age']
    )
)
"""

sorting_list_time = timeit(stmt=sorting_list, globals=globals())
sorting_dict_time = timeit(stmt=sorting_dict, globals=globals())

print(
    f"""\
{sorting_list_time=:.2f} seconds
{sorting_dict_time=:.2f} seconds
list is {(sorting_dict_time/sorting_list_time):.2f} times faster"""
)
```

This code imports the sample data structures for sorting on the `age` attribute. It may seem like you aren't using the imports from `samples`, but it's necessary for these samples to be in the global namespace so that the `timeit` context has access to them.

Running the code for this test on the command line should provide you with some interesting results:

Shell

```
$ python compare_sorting_dict_vs_list.py
sorting_list_time=1.15 seconds
sorting_dict_time=2.26 seconds
list is 1.95 times faster
```

Sorting a list can be almost twice as fast as the process required to sort a dictionary view and then create a new sorted dictionary. So, if you plan on sorting your data very regularly, then a list of tuples might be better than a dictionary for you.

Note: Not many solid conclusions can be drawn from a single dataset like this. Additionally, results can vary wildly with differently sized or shaped data.

These examples are a way for you to dip your toes into the `timeit` module and start to see how and why you might use it. This will give you some of the tools necessary to benchmark your data structures, to help you decide which data structure to settle on for your key-value pairs.

If you need the extra performance, then go ahead and time your specific data structures. That said, beware of premature optimization!

One of the main overheads when sorting a dictionary, as opposed to a list, is reconstructing the dictionary after sorting it. If you were to take out the outer `dict()` constructor, then you'd significantly cut the execution time.

In the next section, you'll be looking at the time it takes to look up values in a dictionary of dictionaries versus in a list of dictionaries.

Comparing the Performance of Lookups

However, if you plan to use the dictionary to sort your data once and use that dictionary mainly for lookups, then a dictionary will definitely make more sense than a list:

Python

```python
# compare_lookup_dict_vs_list.py

from timeit import timeit
from samples import dictionary_of_dictionaries, list_of_dictionaries

lookups = [15, 18, 19, 16, 6, 12, 5, 3, 9, 20, 2, 10, 13, 17, 4, 14, 11, 7, 8]

list_setup = """
def get_key_from_list(key):
    for item in list_of_dictionaries:
        if item["id"] == key:
            return item
"""

lookup_list = """
for key in lookups:
    get_key_from_list(key)
"""

lookup_dict = """
for key in lookups:
    dictionary_of_dictionaries[key]
"""

lookup_list_time = timeit(stmt=lookup_list, setup=list_setup, globals=globals())
lookup_dict_time = timeit(stmt=lookup_dict, globals=globals())

print(
    f"""\
{lookup_list_time=:.2f} seconds
{lookup_dict_time=:.2f} seconds
dict is {(lookup_list_time / lookup_dict_time):.2f} times faster"""
)
```

This code makes a series of lookups to both the list and the dictionary. You'll note that with the list, you have to write a special function to make a lookup. The function to make the list lookup involves going through all the list elements one by one until you find the target element, which isn't ideal.

Running this comparison script from the command line should yield a result showing that dictionary lookups are significantly faster:

Shell

```
$ python compare_lookup_dict_vs_list.py
lookup_list_time=6.73 seconds
lookup_dict_time=0.38 seconds
dict is 17.83 times faster
```

Nearly eighteen times faster! That's a whole bunch. So, you certainly want to weigh the blazing speed of dictionary lookups against the data structure's slower sorting. Bear in mind that this ratio can vary significantly from system to system, not to mention the variation that might come from differently sized dictionaries or lists.

Dictionary lookups are certainly faster, though, no matter how you slice it. That said, if you're just doing lookups, then you could just as easily do that with a regular unsorted dictionary. Why would you need a sorted dictionary in that case? Leave your use case in the comments!

Note: You could try and optimize list lookups, for example by implementing a binary search algorithm to cut time off the list lookup. However, any benefit will only become noticeable at substantial list sizes.

With list sizes like the ones tested here, using a binary search with the `bisect` module is significantly slower than a regular `for` loop.

Now you should have a relatively good idea of some trade-offs between two ways to store your key-value data. The conclusion that you can reach is that, most of the time, if you want a sorted data structure, then you should probably steer clear of the dictionary, mainly for language interoperability reasons.

That said, give Grant Jenks' aforementioned sorted dictionary a try. It uses some ingenious strategies to get around typical performance drawbacks.

Do you have any interesting or performant implementations of a sorted key-value data structure? Share them in the comments, along with your use cases for a sorted dictionary!

Conclusion

You've gone from the most basic way to sort a dictionary to a few advanced alternatives that consider performance in sorting key-value pairs.

In this tutorial, you've:

- Reviewed the `sorted()` function
- Discovered dictionary **views**
- Understood how dictionaries are cast to **lists** during sorting
- Specified **sort keys** to sort a dictionary by value, key, or nested attribute
- Used dictionary **comprehensions** and the `dict()` **constructor** to rebuild your dictionaries
- Considered whether a sorted dictionary is the right **data structure** for your **key-value data**

You're now ready to not only sort dictionaries by any criteria you might think of, but also to judge whether the sorted dictionary is the best choice for you.

Share your sorted dictionary use cases and performance comparisons in the comments below!

Free Download: Click here to download the code that you'll use to sort key-value pairs in this tutorial.

About **Ian Currie**