How to merge two dictionaries in a single expression?

Ask Question

3958

930

I have two Python dictionaries, and I want to write a single expression that returns these two dictionaries, merged. The `update()` method would be what I need, if it returned its result instead of modifying a dict in-place.

```
>>> x = {'a':1, 'b': 2}
>>> y = {'b':10, 'c': 11}
>>> z = x.update(y)
>>> print(z)
None
>>> x
{'a': 1, 'b': 10, 'c': 11}
```

How can I get that final merged dict in `z`, not `x`?

(To be extra-clear, the last-one-wins conflict-handling of `dict.update()` is what I'm looking for as well.)

python dictionary merge

edited Aug 29 '18 at 17:22
wim
172k 56 329 467

asked Sep 2 '08 at 7:44
Carl Meyer
62.1k 16 99 109

39 Answers

1 2 next

4429

How can I merge two Python dictionaries in a single expression?

For dictionaries `x` and `y`, `z` becomes a shallowly merged dictionary with values from `y` replacing those from `x`.

- In Python 3.5 or greater:

```
z = {**x, **y}
```

```
def merge_two_dicts(x, y):
    z = x.copy()   # start wit
    z.update(y)    # modifies
    return z
```

and now:

```
z = merge_two_dicts(x, y)
```

Explanation

Say you have two dicts and you want to merge them into a new dict without altering the original dicts:

```
x = {'a': 1, 'b': 2}
y = {'b': 3, 'c': 4}
```

The desired result is to get a new dictionary (`z`) with the values merged, and the second dict's values overwriting those from the first.

```
>>> z
{'a': 1, 'b': 3, 'c': 4}
```

A new syntax for this, proposed in PEP 448 and available as of Python 3.5, is

```
z = {**x, **y}
```

And it is indeed a single expression.

Note that we can merge in with literal notation as well:

```
z = {**x, 'foo': 1, 'bar': 2, **y}
```

and now:

```
>>> z
{'a': 1, 'b': 3, 'foo': 1, 'bar':
```

It is now showing as implemented in the release schedule for 3.5, PEP 478, and it has now made its way into What's New in Python 3.5 document.

However, since many organizations are still on Python 2, you may wish to do this in a backwards compatible way. The classically Pythonic way, available in Python 2 and Python 3.0-3.4, is to do this as a two-step process:

```
z = x.copy()
z.update(y) # which returns None s
```

`x`'s values, thus `'b'` will point to `3` in our final result.

Not yet on Python 3.5, but want a *single expression*

If you are not yet on Python 3.5, or need to write backward-compatible code, and you want this in a *single expression*, the most performant while correct approach is to put it in a function:

```
def merge_two_dicts(x, y):
    """Given two dicts, merge them
    z = x.copy()
    z.update(y)
    return z
```

and then you have a single expression:

```
z = merge_two_dicts(x, y)
```

You can also make a function to merge an undefined number of dicts, from zero to a very large number:

```
def merge_dicts(*dict_args):
    """
    Given any number of dicts, sha
    precedence goes to key value p
    """
    result = {}
    for dictionary in dict_args:
        result.update(dictionary)
    return result
```

This function will work in Python 2 and 3 for all dicts. e.g. given dicts `a` to `g`:

```
z = merge_dicts(a, b, c, d, e, f,
```

and key value pairs in `g` will take precedence over dicts `a` to `f`, and so on.

Critiques of Other Answers

Don't use what you see in the formerly accepted answer:

```
z = dict(x.items() + y.items())
```

In Python 2, you create two lists in memory for each dict, create a third list in memory with length equal to the length of the first two put together, and then discard all three lists to create the dict. **In Python 3, this will**

```
>>> c = dict(a.items() + b.items()
Traceback (most recent call last):
  File "<stdin>", line 1, in <modu
TypeError: unsupported operand typ
```

and you would have to explicitly create them as lists, e.g. `z = dict(list(x.items()) + list(y.items()))` . This is a waste of resources and computation power.

Similarly, taking the union of `items()` in Python 3 (`viewitems()` in Python 2.7) will also fail when values are unhashable objects (like lists, for example). Even if your values are hashable, **since sets are semantically unordered, the behavior is undefined in regards to precedence. So don't do this:**

```
>>> c = dict(a.items() | b.items()
```

This example demonstrates what happens when values are unhashable:

```
>>> x = {'a': []}
>>> y = {'b': []}
>>> dict(x.items() | y.items())
Traceback (most recent call last):
  File "<stdin>", line 1, in <modu
TypeError: unhashable type: 'list'
```

Here's an example where y should have precedence, but instead the value from x is retained due to the arbitrary order of sets:

```
>>> x = {'a': 2}
>>> y = {'a': 1}
>>> dict(x.items() | y.items())
{'a': 2}
```

Another hack you should not use:

```
z = dict(x, **y)
```

This uses the `dict` constructor, and is very fast and memory efficient (even slightly more-so than our two-step process) but unless you know precisely what is happening here (that is, the second dict is being passed as keyword arguments to the dict constructor), it's difficult to read, it's not the intended usage, and so it is not Pythonic.

Here's an example of the usage being remediated in django.

this method fails in Python 3 when keys are not strings.

```
>>> c = dict(a, **b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <modu
TypeError: keyword arguments must
```

From the mailing list, Guido van Rossum, the creator of the language, wrote:

> I am fine with declaring dict({}, ** {1:3}) illegal, since after all it is abuse of the ** mechanism.

and

> Apparently dict(x, **y) is going around as "cool hack" for "call x.update(y) and return x". Personally I find it more despicable than cool.

It is my understanding (as well as the understanding of the creator of the language) that the intended usage for `dict(**y)` is for creating dicts for readability purposes, e.g.:

```
dict(a=1, b=10, c=11)
```

instead of

```
{'a': 1, 'b': 10, 'c': 11}
```

Response to comments

> Despite what Guido says, `dict(x, **y)` is in line with the dict specification, which btw. works for both Python 2 and 3. The fact that this only works for string keys is a direct consequence of how keyword parameters work and not a short-comming of dict. Nor is using the ** operator in this place an abuse of the mechanism, in fact ** was designed precisely to pass dicts as keywords.

Again, it doesn't work for 3 when keys are non-strings. The implicit calling contract is that namespaces take ordinary dicts, while users must only pass keyword arguments that are strings. All other callables enforced it. `dict` broke this consistency in Python 2.

```
  File "<stdin>", line 1, in <modu
TypeError: foo() keywords must be
>>> dict(**{('a', 'b'): None})
{('a', 'b'): None}
```

This inconsistency was bad given other implementations of Python (Pypy, Jython, IronPython). Thus it was fixed in Python 3, as this usage could be a breaking change.

I submit to you that it is malicious incompetence to intentionally write code that only works in one version of a language or that only works given certain arbitrary constraints.

More comments:

> `dict(x.items() + y.items())` is still the most readable solution for Python 2. Readability counts.

My response: `merge_two_dicts(x, y)` actually seems much clearer to me, if we're actually concerned about readability. And it is not forward compatible, as Python 2 is increasingly deprecated.

> `{**x, **y}` does not seem to handle nested dictionaries. the contents of nested keys are simply overwritten, not merged [...] I ended up being burnt by these answers that do not merge recursively and I was surprised no one mentioned it. In my interpretation of the word "merging" these answers describe "updating one dict with another", and not merging.

Yes. I must refer you back to the question, which is asking for a *shallow* merge of ***two*** dictionaries, with the first's values being overwritten by the second's - in a single expression.

Assuming two dictionary of dictionaries, one might recursively merge them in a single function, but you should be careful not to modify the dicts from either source, and the surest way to avoid that is to make a copy when assigning values. As keys must be hashable and are usually therefore immutable, it is pointless to copy them:

```
    for key in overlapping_keys:
        z[key] = dict_of_dicts_mer
    for key in x.keys() - overlapp
        z[key] = deepcopy(x[key])
    for key in y.keys() - overlapp
        z[key] = deepcopy(y[key])
    return z
```

Usage:

```
>>> x = {'a':{1:{}}, 'b': {2:{}}}
>>> y = {'b':{10:{}}, 'c': {11:{}}
>>> dict_of_dicts_merge(x, y)
{'b': {2: {}, 10: {}}, 'a': {1: {}
```

Coming up with contingencies for other value types is far beyond the scope of this question, so I will point you at my answer to the canonical question on a "Dictionaries of dictionaries merge".

Less Performant But Correct Ad-hocs

These approaches are less performant, but they will provide correct behavior. They will be *much less* performant than `copy` and `update` or the new unpacking because they iterate through each key-value pair at a higher level of abstraction, but they *do* respect the order of precedence (latter dicts have precedence)

You can also chain the dicts manually inside a dict comprehension:

```
{k: v for d in dicts for k, v in d
```

or in python 2.6 (and perhaps as early as 2.4 when generator expressions were introduced):

```
dict((k, v) for d in dicts for k,
```

`itertools.chain` will chain the iterators over the key-value pairs in the correct order:

```
import itertools
z = dict(itertools.chain(x.iterite
```

Performance Analysis

I'm only going to do the performance analysis of the usages known to behave correctly.

```
import timeit
```

In Python 2.7 (system Python):

```
>>> min(timeit.repeat(lambda: merg
0.5726828575134277
>>> min(timeit.repeat(lambda: {k:
1.163769006729126
>>> min(timeit.repeat(lambda: dict
y.iteritems())))
1.1614501476287842
>>> min(timeit.repeat(lambda: dict
d.items())))
2.2345519065856934
```

In Python 3.5 (deadsnakes PPA):

```
>>> min(timeit.repeat(lambda: {**x
0.4094954460160807
>>> min(timeit.repeat(lambda: merg
0.7881555100320838
>>> min(timeit.repeat(lambda: {k:
1.4525277839857154
>>> min(timeit.repeat(lambda: dict
2.3143140770262107
>>> min(timeit.repeat(lambda: dict
d.items())))
3.2069112799945287
```

Resources on Dictionaries

- My explanation of Python's **dictionary implementation**, updated for 3.6.
- Answer on how to add new keys to a dictionary
- Mapping two lists into a dictionary
- The official Python docs on dictionaries
- The Dictionary Even Mightier - talk by Brandon Rhodes at Pycon 2017
- Modern Python Dictionaries, A Confluence of Great Ideas - talk by Raymond Hettinger at Pycon 2017

edited Nov 9 '18 at 2:18

answered Nov 10 '14 at 22:11

Aaron Hall ♦
191k 54 313 266

Strings only limitation for keywords expansion is enough to rule out `{**x, **y}` method. However, the `items` approach can be made workable by converting `dictitems` to `list` like `dict(list(x.items()), list(y.items()))` . – Mohammad Azim May 16 at 15:00

generalized unpacking syntax. To demonstrate that this works: `{**{(0, 1):2}} -> {(0, 1): 2}` – Aaron Hall ♦ May 16 at 16:07

1524

In your case, what you can do is:

```
z = dict(x.items() + y.items())
```

This will, as you want it, put the final dict in `z`, and make the value for key `b` be properly overridden by the second (`y`) dict's value:

```
>>> x = {'a':1, 'b': 2}
>>> y = {'b':10, 'c': 11}
>>> z = dict(x.items() + y.items())
>>> z
{'a': 1, 'c': 11, 'b': 10}
```

If you use Python 3, it is only a little more complicated. To create `z`:

```
>>> z = dict(list(x.items()) + lis
>>> z
{'a': 1, 'c': 11, 'b': 10}
```

edited Aug 29 '18 at 17:18

wim
172k 56 329 467

answered Sep 2 '08 at 7:50

Thomas Vander Stichele
29.7k 11 43 55

589

An alternative:

```
z = x.copy()
z.update(y)
```

answered Sep 2 '08 at 13:00

Matthew Schinckel
28k 4 62 97

66 To clarify why this doesn't meet the critera provided by the question: it's not a single expression and it doesn't return z. – Alex Mar 21 '13 at 13:15

@neuronet every oneliner is usually just moving code that has to happen into a different component and solves it there. this is definitely one of the cases. but other languages have nicer constructs than python for this. and having a referentially transparent variant that returns it's element is a nice to have thing. – Alex Oct 19 '17 at

your code off to...have you really done it in one line? :) I fully agree Python is not good for this: there should be a much easier way. While this answer is more pythonic, is it really all that explicit or clear? `Update` is not one of the "core" functions that people tend to use a lot. – neuronet Oct 19 '17 at 13:07

Another, more concise, option:

301

```
z = dict(x, **y)
```

Note: this has become a popular answer, but it is important to point out that if `y` has any non-string keys, the fact that this works at all is an abuse of a CPython implementation detail, and it does not work in Python 3, or in PyPy, IronPython, or Jython. Also, Guido is not a fan. So I can't recommend this technique for forward-compatible or cross-implementation portable code, which really means it should be avoided entirely.

edited Jan 21 '16 at 6:43

answered Sep 2 '08 at 15:52

Carl Meyer
62.1k 16 99 109

Works fine in Python 3 and PyPy and PyPy 3, can't speak to Jython or Iron. Given this pattern is explicitly documented (see the third constructor form in this documentation) I'd argue it's not an "implementation detail" but intentional feature use. – amcgregor Apr 12 at 13:10

@amcgregor You missed the key phrase "if y has any non-string keys." That's what doesn't work in Python3; the fact that it works in CPython 2 is an implementation detail that can't be relied on. IFF all your keys are guaranteed to be strings, this is a fully supported option. – Carl Meyer May 10 at 16:27

184

This probably won't be a popular answer, but you almost certainly do not want to do this. If you want a copy that's a merge, then use copy (or deepcopy, depending on what you want) and then update. The two lines of code are much more readable

In addition, when you use .items() (pre Python 3.0), you're creating a new list that contains the items from the dict. If your dictionaries are large, then that is quite a lot of overhead (two large lists that will be thrown away as soon as the merged dict is created). update() can work more efficiently, because it can run through the second dict item-by-item.

In terms of time:

```
>>> timeit.Timer("dict(x, **y)", "x
range(1000)))\ny=dict(zip(range(100
15.52571702003479
>>> timeit.Timer("temp = x.copy()\n
range(1000)))\ny=dict(zip(range(100
15.694622993469238
>>> timeit.Timer("dict(x.items() +
range(1000)))\ny=dict(zip(range(100
41.484580039978027
```

IMO the tiny slowdown between the first two is worth it for the readability. In addition, keyword arguments for dictionary creation was only added in Python 2.3, whereas copy() and update() will work in older versions.

edited Aug 5 '14 at 23:56

twasbrillig

7,094 5 27 56

answered Sep 8 '08 at 11:16

Tony Meyer

7,673 5 30 47

130

In a follow-up answer, you asked about the relative performance of these two alternatives:

```
z1 = dict(x.items() + y.items())
z2 = dict(x, **y)
```

On my machine, at least (a fairly ordinary x86_64 running Python 2.5.2), alternative `z2` is not only shorter and simpler but also significantly faster. You can verify this for yourself using the `timeit` module that comes with Python.

Example 1: identical dictionaries mapping 20 consecutive integers to themselves:

```
% python -m timeit -s 'x=y=dict((i,
y.items())'
100000 loops, best of 3: 5.67 usec
% python -m timeit -s 'x=y=dict((i.
```

`z2` wins by a factor of 3.5 or so. Different dictionaries seem to yield quite different results, but `z2` always seems to come out ahead. (If you get inconsistent results for the *same* test, try passing in `-r` with a number larger than the default 3.)

Example 2: non-overlapping dictionaries mapping 252 short strings to integers and vice versa:

```
% python -m timeit -s 'from htmlent
name2codepoint as y' 'z1=dict(x.ite
1000 loops, best of 3: 260 usec per
% python -m timeit -s 'from htmlent
name2codepoint as y' 'z2=dict(x, **
10000 loops, best of 3: 26.9 usec p
```

`z2` wins by about a factor of 10. That's a pretty big win in my book!

After comparing those two, I wondered if `z1` 's poor performance could be attributed to the overhead of constructing the two item lists, which in turn led me to wonder if this variation might work better:

```
from itertools import chain
z3 = dict(chain(x.iteritems(), y.it
```

A few quick tests, e.g.

```
% python -m timeit -s 'from itertoo
codepoint2name as x, name2codepoint
y.iteritems()))'
10000 loops, best of 3: 66 usec per
```

lead me to conclude that `z3` is somewhat faster than `z1` , but not nearly as fast as `z2` . Definitely not worth all the extra typing.

This discussion is still missing something important, which is a performance comparison of these alternatives with the "obvious" way of merging two lists: using the `update` method. To try to keep things on an equal footing with the expressions, none of which modify x or y, I'm going to make a copy of x instead of modifying it in-place, as follows:

```
z0 = dict(x)
z0.update(y)
```

A typical result:

```
% python -m timeit -s 'from htmlent
name2codepoint as y' 'z0=dict(x); z
```

In other words, `z0` and `z2` seem to have essentially identical performance. Do you think this might be a coincidence? I don't....

In fact, I'd go so far as to claim that it's impossible for pure Python code to do any better than this. And if you can do significantly better in a C extension module, I imagine the Python folks might well be interested in incorporating your code (or a variation on your approach) into the Python core. Python uses `dict` in lots of places; optimizing its operations is a big deal.

You could also write this as

```
z0 = x.copy()
z0.update(y)
```

as Tony does, but (not surprisingly) the difference in notation turns out not to have any measurable effect on performance. Use whichever looks right to you. Of course, he's absolutely correct to point out that the two-statement version is much easier to understand.

edited Jan 10 '15 at 2:32

the Tin Man
137k 27 175 258

answered Oct 23 '08 at 2:38

zaphod
3,292 1 19 24

3 This does not work in Python 3; `items()` is not catenable, and `iteritems` does not exist. – Antti Haapala Mar 16 '15 at 5:50

95

I wanted something similar, but with the ability to specify how the values on duplicate keys were merged, so I hacked this out (but did not heavily test it). Obviously this is not a single expression, but it is a single function call.

```
def merge(d1, d2, merge_fn=lambda x
    """
    Merges two dictionaries, non-de
    values on duplicate keys as def
    function.  The default behavior
    with corresponding values in d2
    applicable merge strategy, but
    types in your dicts, so specify
```

```
{'a': 1, 'c': 3, 'b': 2}
>>> merge(d1, d1)
{'a': 1, 'c': 3, 'b': 2}
>>> merge(d1, d1, lambda x,y: x
{'a': 2, 'c': 6, 'b': 4}

"""
result = dict(d1)
for k,v in d2.iteritems():
    if k in result:
        result[k] = merge_fn(re
    else:
        result[k] = v
return result
```

edited Sep 13 '14 at 19:56

Rainy
681 4 11

answered Sep 4 '08 at 19:08

rcreswick
9,648 14 53 67

89

In Python 3, you can use *collections.ChainMap* which groups multiple dicts or other mappings together to create a single, updateable view:

```
>>> from collections import ChainMa
>>> x = {'a':1, 'b': 2}
>>> y = {'b':10, 'c': 11}
>>> z = ChainMap({}, y, x)
>>> for k, v in z.items():
        print(k, '-->', v)

a --> 1
b --> 10
c --> 11
```

edited Sep 27 '14 at 8:12

answered Apr 28 '13 at 3:15

Raymond Hettinger
138k 40 263 357

2 But one should be cautious while using ChainMap there's a catch that if you have duplicate keys the values from first mapping get used and when you call a `del` on say a ChainMap c will delete the first mapping of that key. – Prerit Feb 14 '17 at 5:14

3 @Prerit What else would you expect it to do? That's the normal way chained namespaces work. Consider how $PATH works in bash. Deleting an executable on the path doesn't preclude another executable with the same name further upstream. – Raymond Hettinger Feb 15 '17 at 7:24

69

Recursively/deep update a dict

```
def deepupdate(original, update):
    """
    Recursively update a dict.
    Subdict's won't be overwritten
    """
    for key, value in original.iter
        if key not in update:
            update[key] = value
        elif isinstance(value, dict
            deepupdate(value, updat
    return update
```

Demonstration:

```
pluto_original = {
    'name': 'Pluto',
    'details': {
        'tail': True,
        'color': 'orange'
    }
}

pluto_update = {
    'name': 'Pluutoo',
    'details': {
        'color': 'blue'
    }
}

print deepupdate(pluto_original, pl
```

Outputs:

```
{
    'name': 'Pluutoo',
    'details': {
        'color': 'blue',
        'tail': True
    }
}
```

Thanks rednaw for edits.

edited Dec 18 '15 at 11:19

Dawid Gosławski
1,123 9 22

answered Nov 29 '11 at 11:52

Stan
2,787 1 25 36

This does not answer the question. The question clearly asks for a new dictionary, z, from original dictionaries, x and y, with values from y replacing those of x - not an updated dictionary. This answer modifies y in-place by adding values from x. Worse, it does not copy these values, so one could further modify the modified dictionary, y,

deepcopy to copy the values. – Aaron Hall ♦ Nov 9 '18 at 2:14

@AaronHall agreed this does not answer the question. But it answers my need. I understand those limitations, but that's not an issue in my case. Thinking of it, maybe the name is misleading, as it might evoke a deepcopy, which it does not provide. But it addresses deep nesting. Here's another implementation from the Martellibot: stackoverflow.com/questions/3232943/…. – Jérôme Nov 9 '18 at 13:24

60

The best version I could think while not using copy would be:

```
from itertools import chain
x = {'a':1, 'b': 2}
y = {'b':10, 'c': 11}
dict(chain(x.iteritems(), y.iterite
```

It's faster than `dict(x.items() + y.items())` but not as fast as `n = copy(a); n.update(b)`, at least on CPython. This version also works in Python 3 if you change `iteritems()` to `items()`, which is automatically done by the 2to3 tool.

Personally I like this version best because it describes fairly good what I want in a single functional syntax. The only minor problem is that it doesn't make completely obvious that values from y takes precedence over values from x, but I don't believe it's difficult to figure that out.

answered Oct 14 '10 at 18:55

driax
1,452 15 20

49

```
x = {'a':1, 'b': 2}
y = {'b':10, 'c': 11}
z = dict(x.items() + y.items())
print z
```

For items with keys in both dictionaries ('b'), you can control which one ends up in the output by putting that one last.

answered Sep 2 '08 at 7:49

Greg Hewgill
691k 151 1029 1179

should encapsulate each dict with list() like: dict(list(x.items()) + list(y.items())) – justSaid Apr 26 at 8:45

49

Python 3.5 (PEP 448) allows a nicer syntax option:

```
x = {'a': 1, 'b': 1}
y = {'a': 2, 'c': 2}
final = {**x, **y}
final
# {'a': 2, 'b': 1, 'c': 2}
```

Or even

```
final = {'a': 1, 'b': 1, **x, **y}
```

answered Feb 26 '15 at 21:27

Bilal Syed Hussain
3,041 7 27 39

In what way is this solution better than the `dict(x, **y)` -solution? As you (@CarlMeyer) mentioned within the note of your own answer (stackoverflow.com/a/39858/2798610) Guido considers that solution *illegal*. – Blackeagle52 Mar 4 '15 at 11:09

12 Guido dislikes `dict(x, **y)` for the (very good) reason that it relies on `y` only having keys which are valid keyword argument names (unless you are using CPython 2.7, where the dict constructor cheats). This objection/restriction does not apply to PEP 448, which generalizes the `**` unpacking syntax to dict literals. So this solution has the same concision as `dict(x, **y)`, without the downside. – Carl Meyer Mar 4 '15 at 22:24

40

While the question has already been answered several times, this simple solution to the problem has not been listed yet.

```
x = {'a':1, 'b': 2}
y = {'b':10, 'c': 11}
z4 = {}
z4.update(x)
z4.update(y)
```

It is as fast as z0 and the evil z2 mentioned above, but easy to understand and change.

answered Oct 14 '11 at 16:12

phobie

15:44

14 Yes! The mentioned one-expression-solutions are either slow or evil. Good code is readable and maintainable. So the problem is the question not the answer. We should ask for the best solution of a problem not for a one-line-solution. – phobie Oct 28 '11 at 3:36

6 Lose the `z4 = {}` and change the next line to `z4 = x.copy()` -- better than just good code doesn't do unnecessary things (which makes it even more readable and maintainable). – martineau Mar 8 '13 at 15:10

2 Your suggestion would change this to Matthews answer. While his answer is fine, I think mine is more readable and better maintainable. The extra line would only be bad if it would cost execution time. – phobie May 6 '13 at 11:50

38

```
def dict_merge(a, b):
  c = a.copy()
  c.update(b)
  return c

new = dict_merge(old, extras)
```

Among such shady and dubious answers, this shining example is the one and only good way to merge dicts in Python, endorsed by dictator for life *Guido van Rossum* himself! Someone else suggested half of this, but did not put it in a function.

```
print dict_merge(
      {'color':'red', 'model':'Mini
      {'model':'Ferrari', 'owner':'
```

gives:

```
{'color': 'red', 'owner': 'Carl', '
```

edited Aug 6 '12 at 9:30

answered Aug 6 '12 at 9:24

Sam Watkins
4,815 2 26 29

32

If you think lambdas are evil then read no further. As requested, you can write the fast and memory-efficient solution with one expression:

```
{'a': 1, 'c': 11, 'b': 10}
print x
{'a': 1, 'b': 2}
```

As suggested above, using two lines or writing a function is probably a better way to go.

edited Nov 23 '11 at 18:20

answered Nov 23 '11 at 18:08

EMS
578 1 6 11

26

Be pythonic. Use a comprehension:

```
z={i:d[i] for d in [x,y] for i in d
```

```
>>> print z
{'a': 1, 'c': 11, 'b': 10}
```

edited Sep 29 '16 at 10:45

answered Jan 20 '16 at 11:46

Robino
2,335 1 20 32

1 As a function: `def dictmerge(*args): return {i:d[i] for d in args for i in d}` – Jesse Knight Jun 6 '18 at 18:27

1 Save a lookup by iterating the key/value pairs directly: `z={k: v for d in (x, y) for k, v in d.items()}` – ShadowRanger Mar 5 at 19:33

23

In python3, the `items` method no longer returns a list, but rather a *view*, which acts like a set. In this case you'll need to take the set union since concatenating with `+` won't work:

```
dict(x.items() | y.items())
```

For python3-like behavior in version 2.7, the `viewitems` method should work in place of `items`:

```
dict(x.viewitems() | y.viewitems())
```

I prefer this notation anyways since it seems more natural to think of it as a

Edit:

A couple more points for python 3. First, note that the `dict(x, **y)` trick won't work in python 3 unless the keys in `y` are strings.

Also, Raymond Hettinger's Chainmap answer is pretty elegant, since it can take an arbitrary number of dicts as arguments, but from the docs it looks like it sequentially looks through a list of all the dicts for each lookup:

> Lookups search the underlying mappings successively until a key is found.

This can slow you down if you have a lot of lookups in your application:

```
In [1]: from collections import Cha
In [2]: from string import ascii_up
dict(zip(lo, up)); y = dict(zip(up,
In [3]: chainmap_dict = ChainMap(y,
In [4]: union_dict = dict(x.items()
In [5]: timeit for k in union_dict:
100000 loops, best of 3: 2.15 µs pe
In [6]: timeit for k in chainmap_di
10000 loops, best of 3: 27.1 µs per
```

So about an order of magnitude slower for lookups. I'm a fan of Chainmap, but looks less practical where there may be many lookups.

edited May 23 '17 at 12:34
Community ♦
1 1

answered Oct 9 '13 at 18:09
beardc
8,474 12 57 75

18

Abuse leading to a one-expression solution for Matthew's answer:

```
>>> x = {'a':1, 'b': 2}
>>> y = {'b':10, 'c': 11}
>>> z = (lambda f=x.copy(): (f.upda
>>> z
{'a': 1, 'c': 11, 'b': 10}
```

You said you wanted one expression, so I abused `lambda` to bind a name, and tuples to override lambda's one-expression limit. Feel free to cringe.

You could also do this of course if you don't care about copying it:

```
>>> z
{'a': 1, 'b': 10, 'c': 11}
```

edited May 23 '17 at 12:34

Community ♦
1 1

answered Aug 7 '13 at 21:23

Claudiu
130k 128 403 598

18

Simple solution using itertools that preserves order (latter dicts have precedence)

```
import itertools as it
merge = lambda *args: dict(it.chain
args)))
```

And it's usage:

```
>>> x = {'a':1, 'b': 2}
>>> y = {'b':10, 'c': 11}
>>> merge(x, y)
{'a': 1, 'b': 10, 'c': 11}

>>> z = {'c': 3, 'd': 4}
>>> merge(x, y, z)
{'a': 1, 'b': 10, 'c': 3, 'd': 4}
```

edited Sep 6 '16 at 11:30

answered Aug 4 '15 at 14:54

reubano
2,610 21 30

17

Two dictionaries

```
def union2(dict1, dict2):
    return dict(list(dict1.items())
```

***n* dictionaries**

```
def union(*dicts):
    return dict(itertools.chain.fro
```

`sum` has bad performance. See https://mathieularose.com/how-not-to-flatten-a-list-of-lists-in-python/

edited Oct 2 '16 at 18:16

answered Oct 17 '12 at 2:09

13

Even though the answers were good for this *shallow* dictionary, none of the methods defined here actually do a deep dictionary merge.

Examples follow:

```
a = { 'one': { 'depth_2': True }, '
b = { 'one': { 'extra': False } }
print dict(a.items() + b.items())
```

One would expect a result of something like this:

```
{ 'one': { 'extra': False', 'depth_
```

Instead, we get this:

```
{'two': True, 'one': {'extra': Fals
```

The 'one' entry should have had 'depth_2' and 'extra' as items inside its dictionary if it truly was a merge.

Using chain also, does not work:

```
from itertools import chain
print dict(chain(a.iteritems(), b.i
```

Results in:

```
{'two': True, 'one': {'extra': Fals
```

The deep merge that rcwesick gave also creates the same result.

Yes, it will work to merge the sample dictionaries, but none of them are a generic mechanism to merge. I'll update this later once I write a method that does a true merge.

answered Aug 3 '12 at 23:36

Thanh Lim

155 1 2

This is a commentary on the existing answers, and does not provide an answer. – cpburnz Apr 5 '18 at 15:41

10

Drawing on ideas here and elsewhere I've comprehended a function:

```
def merge(*dicts, **kv):
      return { k:v for d in list(di
```

Usage (tested in python 3):

```
assert (merge(foo='bar')=={'foo': '

assert (merge({1:11},{1:99},foo='ba
    {1: 99, 'foo': 'bar', 'baz':'qu

assert (merge({1:11},{1:99})=={1: 9
```

You could use a lambda instead.

answered Jul 19 '13 at 5:49

Bijou Trouvaille

3,450 3 30 39

9

The problem I have with solutions listed to date is that, in the merged dictionary, the value for key "b" is 10 but, to my way of thinking, it should be 12. In that light, I present the following:

```
import timeit

n=100000
su = """
x = {'a':1, 'b': 2}
y = {'b':10, 'c': 11}
"""

def timeMerge(f,su,niter):
    print "{:4f} sec for: {:30s}".f

timeMerge("dict(x, **y)",su,n)
timeMerge("x.update(y)",su,n)
timeMerge("dict(x.items() + y.items
timeMerge("for k in y.keys(): x[k]

#confirm for loop adds b entries to
x = {'a':1, 'b': 2}
y = {'b':10, 'c': 11}
for k in y.keys(): x[k] = k in x an
print "confirm b elements are added
```

Results:

```
0.049465 sec for: dict(x, **y)
0.033729 sec for: x.update(y)
0.150380 sec for: dict(x.items() +
0.083120 sec for: for k in y.keys()

confirm b elements are added: {'a':
```

answered Dec 3 '13 at 18:11

upandacross

192 1 6

You might be interested in `cytoolz.merge_with` (toolz.readthedocs.io/en/latest/…) – bli Feb 16 '17 at 13:44

```
>>> x = {'a':1, 'b': 2}
```

```
>>> y
{'c': 11, 'b': 10}
>>> z
{'a': 1, 'c': 11, 'b': 10}
```

answered Nov 13 '13 at 10:01

John La Rooy
218k 41 284 434

This method overwrites `x` with its copy. If `x` is a function argument this won't work (see example) – bartolo-otrit Feb 22 at 9:27

8

This can be done with a single dict comprehension:

```
>>> x = {'a':1, 'b': 2}
>>> y = {'b':10, 'c': 11}
>>> { key: y[key] if key in y else
      for key in set(x) + set(y)
    }
```

In my view the best answer for the 'single expression' part as no extra functions are needed, and it is short.

answered Jul 17 '15 at 14:47

RemcoGerlich
20.5k 3 37 58

I suspect performance will not be very good though; creating a set out of each dict then only iterating through the keys means another lookup for the value each time (though relatively fast, still increases the order of the function for scaling) – Breezer Feb 16 '17 at 14:57

2 it all depends on the version of the python we are using. In 3.5 and above {**x,**y} gives the concatenated dictionary – Rashid Mv Dec 23 '17 at 15:50

8

```
from collections import Counter
dict1 = {'a':1, 'b': 2}
dict2 = {'b':10, 'c': 11}
result = dict(Counter(dict1) + Coun
```

This should solve your problem.

answered Nov 30 '15 at 13:04

reetesh11
305 4 12

(For Python2.7* only: there are simpler

If you're not averse to importing a standard library module, you can do

```
from functools import reduce

def merge_dicts(*dicts):
    return reduce(lambda a, d: a.up
```

(The `or a` bit in the `lambda` is necessary because `dict.update` always returns `None` on success.)

answered Mar 28 '16 at 13:13

kjo
10.9k 30 98 188

6

It's so silly that `.update` returns nothing.
I just use a simple helper function to solve the problem:

```
def merge(dict1,*dicts):
    for dict2 in dicts:
        dict1.update(dict2)
    return dict1
```

Examples:

```
merge(dict1,dict2)
merge(dict1,dict2,dict3)
merge(dict1,dict2,dict3,dict4)
merge({},dict1,dict2)  # this one r
```

answered Mar 2 '14 at 1:44

GetFree
23.4k 15 61 84

4

Using a dict comprehension, you may

```
x = {'a':1, 'b': 2}
y = {'b':10, 'c': 11}

dc = {xi:(x[xi] if xi not in list(y
           else y[xi]) for xi in li
```

gives

```
>>> dc
{'a': 1, 'c': 11, 'b': 10}
```

Note the syntax for `if else` in comprehension

```
{ (some_key if condition else defau
          else something_if_false)
```

8 I like the idea of using a dict comprehension, but your implementation is weak. It is insane to use `... in list(y.keys())` instead of just `... in y` . – wim Feb 18 '14 at 20:18

4

I know this does not really fit the specifics of the questions ("one liner"), but since *none* of the answers above went into this direction while lots and lots of answers addressed the performance issue, I felt I should contribute my thoughts.

Depending on the use case it might not be necessary to create a "real" merged dictionary of the given input dictionaries. A *view* which does this might be sufficient in many cases, i. e. an object which acts *like* the merged dictionary would without computing it completely. A lazy version of the merged dictionary, so to speak.

In Python, this is rather simple and can be done with the code shown at the end of my post. This given, the answer to the original question would be:

```
z = MergeDict(x, y)
```

When using this new object, it will behave like a merged dictionary but it will have constant creation time and constant memory footprint while leaving the original dictionaries untouched. Creating it is way cheaper than in the other solutions proposed.

Of course, if you use the result a lot, then you will at some point reach the limit where creating a real merged dictionary would have been the faster solution. As I said, it depends on your use case.

If you ever felt you would prefer to have a real merged `dict` , then calling `dict(z)` would produce it (but way more costly than the other solutions of course, so this is just worth mentioning).

You can also use this class to make a kind of copy-on-write dictionary:

```
a = { 'x': 3, 'y': 4 }
b = MergeDict(a)  # we merge just o
b['x'] = 5
print b  # will print {'x': 5, 'y':
```

Here's the straight-forward code of `MergeDict`:

```
class MergeDict(object):
  def __init__(self, *originals):
    self.originals = ({},) + origin

  def __getitem__(self, key):
    for original in self.originals:
      try:
        return original[key]
      except KeyError:
        pass
    raise KeyError(key)

  def __setitem__(self, key, value)
    self.originals[0][key] = value

  def __iter__(self):
    return iter(self.keys())

  def __repr__(self):
    return '%s(%s)' % (
      self.__class__.__name__,
      ', '.join(repr(original)
          for original in reversed(

  def __str__(self):
    return '{%s}' % ', '.join(
        '%r: %r' % i for i in self.

  def iteritems(self):
    found = set()
    for original in self.originals:
      for k, v in original.iteritem
        if k not in found:
          yield k, v
          found.add(k)

  def items(self):
    return list(self.iteritems())

  def keys(self):
    return list(k for k, _ in self.

  def values(self):
    return list(v for _, v in self.
```

answered May 18 '16 at 15:57

Alfe
34.1k 10 66 113

2 I saw by now that some answers refer to a class called `ChainMap` which is available in Python 3 only and which does more or less what my code does. So shame on me for not reading everything carefully enough. But given that this only exists for Python 3, please take my answer as a contribution for the Python 2 users ;-) – Alfe May 18 '16 at 16:10

4 ChainMap was backported for earlier Pythons: pypi.python.org/pypi/chainmap – clacke Jul 28 '16 at 11:19

1 2 next