

Hashing a dictionary?

[Ask Question](#)

For caching purposes I need to generate a cache key from GET arguments which are present in a dict.

Currently I'm using `sha1(repr(sorted(my_dict.items())))` (`sha1()` is a convenience method that uses `hashlib` internally) but I'm curious if there's a better way.

[python](#)[hash](#)[dictionary](#)

edited May 20 '17 at 22:05



[martineau](#)

63k 8 87 167

asked May 4 '11 at 13:19



[ThiefMaster](#) ♦

233k 59 457 549

That seems good to me. – [Devin Jeanpierre](#)
May 4 '11 at 13:22

3 this might not work with nested dict. shortest solution is to use `json.dumps(my_dict, sort_keys=True)` instead, which will recurse into dict values. – [Andrey Fedorov](#) Apr 8 '14 at 19:15

FYI re: `dumps`, stackoverflow.com/a/12739361/1082367 says "The output from `pickle` is not guaranteed to be canonical for similar

for hashing." –

[Matthew Cornell](#) Dec

16 '14 at 12:49

sort the dict keys, not the items, i would also send the keys to the hash function. – [nyuwec](#) May 26 '16 at 12:59

- 1 Interesting backstory about hashing mutable data structures (like dictionaries): python.org/dev/peps/pep-0351 was proposed to allow arbitrarily freezing objects, but rejected. For rationale, see this thread in python-dev: mail.python.org/pipermail/python-dev/2006-February/060793.html – [FluxLemur](#) Mar 29 at 16:47
-

9 Answers

If your dictionary is not nested, you could make a frozenset with the dict's items and use [hash\(\)](#) :

```
hash(frozenset(my_dict.items()))
```

This is much less computationally intensive than generating the JSON string or representation of the dictionary.

edited Apr 8 '15 at 11:39



[Quentin Pradet](#)

3,361 1 20 39

answered May 4 '11 at 13:24



[Imran](#)

42.3k 19 85 116

complicated). The OP's solution works perfectly fine. I substituted sha1 with hash to save an import. – [spatel](#) Jan 18 '12 at 7:51

8 @Ceaser That won't work because tuple implies ordering but dict items are unordered. frozenset is better. – [Antimony](#) Jul 30 '12 at 11:55


16 Beware of the built-in hash if something needs to be consistent across different machines. Implementations of python on cloud platforms like Heroku and GAE will return different values for hash() on different instances making it useless for anything that must be shared between two or more "machines" (dynos in the case of heroku) – [Ben Roberts](#) Apr 22 '15 at 22:40

2 It might be interesting the hash() function does not produce a stable output. This means that, given the same input, it returns

Join Stack Overflow to learn, share knowledge, and build your career.

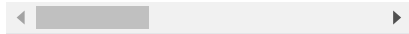
By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

Sign Up 

every time the interpreter is started. – [Hermann Schachner](#) May 28 '15 at 11:03 

4 expected. the seed is introduced for security reason as far as I remember to add some kind of memory

Nikokrock May 29 '15
at 13:03



EDIT: If *all your keys are strings*, then before continuing to read this answer, please see Jack O'Connor's significantly [simpler \(and faster\) solution](#) (which also works for hashing nested dictionaries).

Although an answer has been accepted, the title of the question is "Hashing a python dictionary", and the answer is incomplete as regards that title. (As regards the body of the question, the answer is complete.)

Nested Dictionaries

If one searches Stack Overflow for how to hash a dictionary, one might stumble upon this aptly titled question, and leave unsatisfied if one is attempting to hash multiply nested dictionaries. The answer above won't work in this case, and you'll have to implement some sort of recursive mechanism to retrieve the hash.

Here is one such mechanism:

```
import copy

def make_hash(o):
    """
```

```

if isinstance(o, (set, frozenset)):
    return tuple([make_hash(v) for v in o])

elif not isinstance(o, dict):
    return hash(o)

new_o = copy.deepcopy(o)
for k, v in new_o.items():
    new_o[k] = make_hash(v)

return hash(tuple(frozenset(new_o.items())))

```

Bonus: Hashing Objects and Classes

The hash() function works great when you hash classes or instances. However, here is one issue I found with hash, as regards objects:

```

class Foo(object): pass
foo = Foo()
print (hash(foo)) # 12098:
foo.a = 1
print (hash(foo)) # 12098:

```

The hash is the same, even after I've altered foo. This is because the identity of foo hasn't changed, so the hash is the same. If you want foo to hash differently depending on its current definition, the solution is to hash off whatever is actually changing. In this case, the `__dict__` attribute:

```

class Foo(object): pass
foo = Foo()
print (make_hash(foo.__dict__))
foo.a = 1
print (make_hash(foo.__dict__))

```

Alas, when you attempt to do the same thing with the class itself:

```
print (type(Foo.__dict__))
```

Here is a similar mechanism as previous that will handle classes appropriately:

```
import copy

DictProxyType = type(object.__dict__)

def make_hash(o):
    """
    Makes a hash from a dict (which
    contains only other hashable
    dictionaries). In the case of a
    to be hashed, pass in a dict.
    For example, a class can be
    hashed by passing in its __dict__.

    make_hash([cls.__dict__])

    A function can be hashed by passing
    its __dict__ to make_hash.

    make_hash([fn.__dict__])
    """
    if type(o) == DictProxyType:
        o2 = {}
        for k, v in o.items():
            if not k.startswith('__'):
                o2[k] = v
        o = o2

    if isinstance(o, (set, tuple)):
        return tuple([make_hash(v) for v in o])

    elif not isinstance(o, DictProxyType):
        return hash(o)

    new_o = copy.deepcopy(o)
    for k, v in new_o.items():
        new_o[k] = make_hash(v)

    return hash(tuple(frozenset(new_o.items())))
```

You can use this to return a hash tuple of however many elements you'd like:

```
# -7666086133114527897
print (make_hash(func.__dict__))

# (-7666086133114527897, ...)
print (make_hash([func.__dict__]))
```

NOTE: all of the above code assumes Python 3.x. Did not test in earlier versions, although I assume `make_hash()` will work in, say, 2.7.2. As far as making the examples work, I *do* know that

```
func.__code__
```

should be replaced with

```
func.func_code
```

edited Sep 26 '17 at 16:40



[MByD](#)

115k 22 220 247

answered Jan 3 '12 at 15:05



[jomido](#)

840 10 16

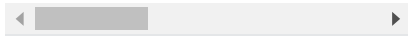
isinstance takes a sequence for the second argument, so `isinstance(o, (set, tuple, list))` would work. – [Xealot](#) Feb 13 '13 at 1:42

@Xealot Great thanks for that. Updated. – [jomido](#) Feb 13 '13 at 13:50

thanks for making me realize frozenset could consistently hash querystring parameters :) – [Xealot](#) Feb 14 '13 at 14:26

-
- 1 The items need to be sorted in order to create the same hash if the dict item order is different but the key values aren't -> return `hash(tuple(frozenset(sorted(new_o.items()))))` – [Bas Koopmans](#) Oct 28

and tuples. Otherwise it takes my lists of integers that happen to be values in my dictionary, and returns back lists of hashes, which is not what I want. – [osa](#) Jan 7 '15 at 23:09



The code below avoids using the Python `hash()` function because it will not provide hashes that are consistent across restarts of Python (see [hash function in Python 3.3 returns different results between sessions](#)).

`make_hashable()` will convert the object into nested tuples and `make_hash_sha256()` will also convert the `repr()` to a base64 encoded SHA256 hash.

```
import hashlib
import base64

def make_hash_sha256(o):
    hasher = hashlib.sha256()
    hasher.update(repr(o).encode('utf-8'))
    return base64.b64encode(hasher.digest()).decode('utf-8')

def make_hashable(o):
    if isinstance(o, (tuple, list)):
        return tuple((make_hashable(i) if isinstance(i, dict) else i) for i in o)

    if isinstance(o, dict):
        return tuple(sorted((make_hashable(k), make_hashable(v)) for k, v in o.items()))

    if isinstance(o, (set, frozenset)):
        return tuple(sorted((make_hashable(i) if isinstance(i, dict) else i) for i in o))

    return o

o = dict(x=1, b=2, c=[3, 4, 5])
print(make_hashable(o))
# (('b', 2), ('c', (3, 4, 5)), ('x', 1))
```


edited May 23 '17 at 11:33



Community ♦

1 1

answered Feb 10 '17 at 5:09



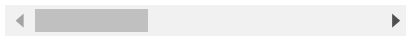
Claudio Fahey

168 1 5

```

make_hash_sha256((
    (0, 1),
    (2, 3))) == make_hash_
sha256({0: 1, 2: 3}) ==
make_hash_sha256({2
: 3, 0: 1}) != make_hash
_sha256(((2, 3),
(0, 1))) . This isn't
quite the solution I'm
looking for, but it is a
nice intermediate. I'm
thinking of adding
type(o).__name__ to
the beginning of each of
the tuples to force
differentiation. – Poik
Sep 17 '17 at 20:01

```



Updated from 2013 reply...

None of the above answers seem reliable to me. The reason is the use of `items()`. As far as I know, this comes out in a machine-dependent order.

How about this instead?

```

import hashlib

def dict_hash(the_dict, *ignore):
    if ignore: # Sometimes
        interesting = the_dict
        for item in ignore:
            if item in interesting:
                interesting = the_dict - item
        the_dict = interesting
    result = hashlib.sha1(
        '%s' % sorted(the_dict.items())
    ).hexdigest()
    return result

```

answered Mar 4 '13 at 18:10



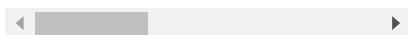
[Steve Yeago](#)

321 3 8

Why do you think it matters that `dict.items` does not return a predictably ordered list? `frozenset` takes care of that – [glarrai](#) Jul 11 '14 at 22:54

- 2 A set, by definition, is unordered. Thus the order in which objects are added is irrelevant. You do have to realize that the built-in function `hash` does not care about how the `frozenset` contents are printed or something like that. Test it in several machines and python versions and you'll see. – [glarrai](#) Jul 13 '14 at 3:38
-

Why do you use the extra `hash()` call in `value = hash('%s::%s' % (value, type(value)))??` – [RuiDo](#) Jul 6 '16 at 10:12



Using `sorted(d.items())` isn't enough to get us a stable repr. Some of the values in `d` could be dictionaries too, and their keys will still come out in an arbitrary order. As long as all the keys are strings, I prefer to use:

```
json.dumps(d, sort_keys=True)
```

Python versions, I'm not certain that this is bulletproof. You might want to add the `separators` and `ensure_ascii` arguments to protect yourself from any changes to the defaults there. I'd appreciate comments.

edited Jun 10 '16 at 14:24

answered Feb 25 '14 at 2:29



[Jack O'Connor](#)

4,569 1 25 33

-
- 1 This seems like the best solution, but could you expound on why you think `separators` and `ensure_ascii` might be useful? – [Andrey Fedorov](#) Apr 8 '14 at 19:13
-
- 3 I tested the performance of this with different dataset, it's much much faster than `make_hash` .
gist.github.com/charlax/b8731de51d2ea86c6eb9 – [charlax](#) Sep 18 '14 at 22:33
-
- 2 @charlax `ujson` doesn't guarantee the order of the dict pairs, so it's not safe to do that – [arthurprs](#) Jul 3 '15 at 12:48
-
- 6 This solution only works as long as all keys are strings, e.g. `json.dumps({'a': {(0, 5): 5, 1: 3}})` fails. – [kadee](#) Jun 1 '16 at 11:01
-
- 2 Some datatype are not json serializable like

To preserve key order,
instead of

```
hash(str(dictionary)) or
hash(json.dumps(dictionary))
```

I would prefer quick-and-dirty solution:

```
from pprint import pformat
h = hash(pformat(dictionary))
```

It will work even for types
like `DateTime` and more
that are not JSON
serializable.

answered Jan 30 '15 at 0:45



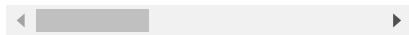
[shirk3y](#)

134 3

3 Who guarantees that
pformat or json always
use the same order? –
[ThiefMaster ♦](#) Jan 30
'15 at 6:02

1 @ThiefMaster,
"Changed in version
2.5: Dictionaries are
sorted by key before the
display is computed;
before 2.5, a dictionary
was sorted only if its
display required more
than one line, although
that wasn't
documented."
docs.python.org/2/library/pprint.html) – [Arel](#)
Jan 15 '16 at 16:21

This doesn't seem valid
to me. The pprint
modules and pformat
are understood by the
authors to be for display
purposes and not
serialization. Because
of this, you shouldn't
feel safe in assuming
that pformat will always



I do it like this:

```
hash(str(my_dict))
```

answered Dec 26 '14 at 23:53



[garbanzio](#)

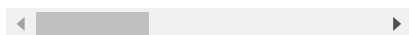
420 1 4 9

1 Can someone explain what is so wrong with this method? – [mhristache](#) Oct 20 '16 at 12:32

4 @maximi Dictionaries are not stable in terms of order, thus

```
hash(str({'a': 1, 'b': 2})) != hash(str({'b': 2, 'a': 1}))
```

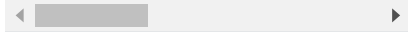
 (while it might work for some dictionaries, it is not guaranteed to work on all). – [Vlad Frolov](#) Mar 1 '17 at 9:48 ✎



Here is a clearer solution.

```
def freeze(o):
    if isinstance(o, dict):
        return frozenset({ k: freeze(v) for k, v in o.items() })
    elif isinstance(o, list):
        return tuple([freeze(o) for o in o])
    return o

def make_hash(o):
    """
    makes a hash out of an object, including string and numerical values.
    """
    return hash(freeze(o))
```



The general approach is fine, but you may want to consider the hashing method.

SHA was designed for cryptographic strength (speed too, but strength is more important). You may want to take this into account. Therefore, using the built-in `hash` function is probably a good idea, unless security is somehow key here.

answered May 4 '11 at 13:24



[Eli Bendersky](#)

159k 63 292 363

-
- 6 built-in `hash` function is not designed to store the computed value, and hash result may vary with different versions of python. — [Taha Jahangir](#) Jan 18 '13 at 7:16
-