How to retrieve an element from a set without removing it?

Asked 16 years, 3 months ago Modified 3 years, 6 months ago Viewed 963k times

707

Suppose the following:

```
>>> s = set([1, 2, 3])
```

How do I get a value (any value) out of `s` without doing `s.pop()`? I want to leave the item in the set until I am sure I can remove it - something I can only be sure of after an asynchronous call to another host.

Quick and dirty:

```
>>> elem = s.pop()
>>> s.add(elem)
```

But do you know of a better way? Ideally in constant time.

python set

Share

Improve this question

Follow

edited Feb 19, 2018 at 22:22

MSeifert

152k ● 41 ● 349 ● 366

asked Sep 12, 2008 at 19:58

Daren Thomas

70.2k ● 42 ● 155 ● 205

43 Anyone know why python doesn't already have this function implemented? – hlin117 Jun 22, 2015 at 7:50

7 What's the use case? Set doesn't have this ability for a reason. You supposed to iterate through it and make set related operations like `union` etc not taking elements from it. For example `next(iter({3,2,1}))` always returns `1` so if you thought that this would return random element - it wouldn't. So maybe you just using the wrong data structure? What's the use case? – user1685095 May 28, 2017 at 13:59

1 Related: stackoverflow.com/questions/20625579/... (I know, it's not the same question, but there are worthwhile alternatives and insights there.) – John Y Nov 13, 2017 at 19:47

2 @hlin117 So why does this then make no sense? It is called "drawing with replacement"... – Radio Controlled Nov 20, 2020 at 7:36

5 One reasonable use case that I keep encountering is this: I am writing a test, and I get a set. I want to look at any value in it to build more data for the test. I don't care which one I get, and I

don't really care if its the same or different each time. I just need *a* value from the set. – Troy Daniels Dec 3, 2021 at 18:33

15 Answers

Sorted by: Highest score (default)

871

Two options that don't require copying the whole set:

```
for e in s:
    break
# e is now an element from s
```

Or...

```
e = next(iter(s))
```

But in general, sets don't support indexing or slicing.

Share

Improve this answer

Follow

edited Mar 11, 2014 at 6:56

Raymond Hettinger

226k ● 66 ● 400 ● 500

answered Sep 12, 2008 at 20:08

Blair Conrad

241k ● 25 ● 136 ● 112

4 This answers my question. Alas, I guess I will still use pop(), since iteration seems to sort the elements. I would prefer them in random order... – Daren Thomas Sep 12, 2008 at 20:17

22 I don't think that the iter() is sorting the elements - when I create a set and pop() until it's empty, I get consistent (sorted, in my example) ordering, and it's the same as the iterator - pop() doesn't promise random order, just arbitrary, as in "I promise nothing". – Blair Conrad Sep 12, 2008 at 20:20

10 +1 `iter(s).next()` is not gross but great. Completely general to take arbitrary element from any iterable object. Your choice if you want to be careful if the collection is empty though. – u0b34a0f6ae Oct 23, 2009 at 13:25

20 next(iter(s)) is also OK and I tend to think it reads better. Also, you can use a sentinel to handle the case when s is empty. E.g. next(iter(s), set()). – j-a Jul 22, 2012 at 6:34

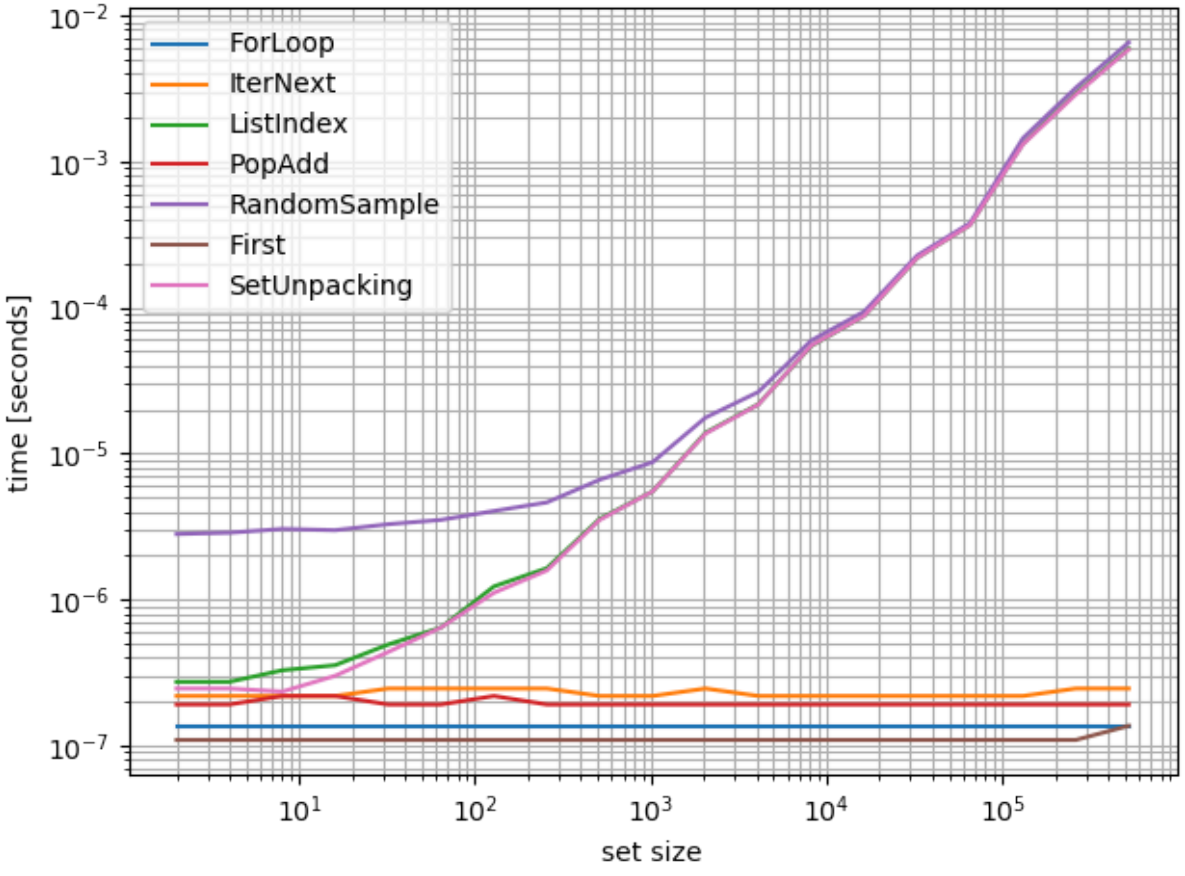
21 `next(iter(your_list or []), None)` to handle None sets and empty sets – MrE Jul 31, 2018 at 14:22

227

I wondered how the functions will perform for different sets, so I did a benchmark:

```
from random import sample

def ForLoop(s):
    for e in s:
        break
```

```
    return e

def IterNext(s):
    return next(iter(s))

def ListIndex(s):
    return list(s)[0]

def PopAdd(s):
    e = s.pop()
    s.add(e)
    return e

def RandomSample(s):
    return sample(s, 1)

def SetUnpacking(s):
    e, *_ = s
    return e

from simple_benchmark import benchmark

b = benchmark([ForLoop, IterNext, ListIndex, PopAdd, RandomSample,
SetUnpacking],
              {2**i: set(range(2**i)) for i in range(1, 20)},
              argument_name='set size',
              function_aliases={first: 'First'})

b.plot()
```



This plot clearly shows that some approaches (`RandomSample`, `SetUnpacking` and `ListIndex`) depend on the size of the set and should be avoided in the general case (at least if performance *might* be important). As already shown by the other answers the fastest way is `ForLoop`.

However as long as one of the constant time approaches is used the performance difference will be negligible.

`iteration_utilities` (Disclaimer: I'm the author) contains a convenience function for this use-case: `first`:

```
>>> from iteration_utilities import first
>>> first({1,2,3,4})
1
```

I also included it in the benchmark above. It can compete with the other two "fast" solutions but the difference isn't much either way.

Share

Improve this answer

Follow

edited Apr 9, 2018 at 21:14

answered Feb 19, 2018 at 21:52

MSeifert

152k ● 41 ● 349 ● 366

4 This is a great answer. Thanks for putting in the time to make it empirical. – Eric McLachlan Mar 21, 2021 at 6:50

graph gives more attention to answer – DikShU Jul 3, 2021 at 12:40

1 I have a short question, why do you use break in the ForLoop instead of using `return e` directly? The function should "break" the moment the return is executed. – Andreas Sep 21, 2021 at 14:55

@Andreas That's a good and valid point. Thanks for bringing it up. But for the "why": I wanted to compare the runtime from the other answers so I simply copied the approach from those. In this case the answer had the `break` (ref stackoverflow.com/a/59841)... not a good answer but I simply didn't want to change their code too much. – MSeifert Sep 23, 2021 at 9:36

1 @DanielJerrehian In that case you can provide a default value `first(set(), default=None)` for example :) – MSeifert Dec 14, 2021 at 18:01

214

Least code would be:

```
>>> s = set([1, 2, 3])
>>> list(s)[0]
1
```

Obviously this would create a new list which contains each member of the set, so not great if your set is very large.

Share Improve this answer Follow

answered Sep 13, 2008 at 1:07

John

15.3k ● 12 ● 59 ● 57

33 @augurar: Because it gets the job done in a relatively simple manner. And sometimes that's all that matters in a quick script. – tonysdg Feb 21, 2017 at 1:58

5 @augurar I think people voted on this answer because `set` is not made for indexing and slicing primarily; and this user just shifted the coder to use the suitable datatype for such work i.e. `list`. – Vicrobot Jul 14, 2018 at 15:26

10 @Vicrobot Yeah but it does so by copying the entire collection and turning an O(1) operation into an O(n) operation. This is a terrible solution that no one should ever use. – augurar Jul 16, 2018 at 22:01

26 Also if you're just aiming for "least code" (which is dumb), then `min(s)` uses even fewer characters while being just as terrible and inefficient as this. – augurar Jul 16, 2018 at 22:06

18 +1 for the code golf winner, which I have a practical counterexample for being "terrible and inefficient": `min(s)` is slightly faster than `next(iter(s))` for sets of size 1, and I came to this answer specifically looking to special-case extracting the only element from sets of size 1. – lehiester Jan 18, 2019 at 19:49

74

tl;dr

`for first_item in muh_set: break` remains the optimal approach in Python 3.x.
Curse you, Guido.

y u do this

Welcome to yet another set of Python 3.x timings, extrapolated from wr.'s excellent Python 2.x-specific response. Unlike AChampion's equally helpful Python 3.x-specific response, the timings below *also* time outlier solutions suggested above – including:

- `list(s)[0]`, John's novel sequence-based solution.
- `random.sample(s, 1)`, dF.'s eclectic RNG-based solution.

Code Snippets for Great Joy

Turn on, tune in, time it:

```
from timeit import Timer

stats = [
    "for i in range(1000): \n\tfor x in s: \n\t\tbreak",
    "for i in range(1000): next(iter(s))",
    "for i in range(1000): s.add(s.pop())",
    "for i in range(1000): list(s)[0]",
    "for i in range(1000): random.sample(s, 1)",
]

for stat in stats:
    t = Timer(stat, setup="import random\ns=set(range(100))")
    try:
        print("Time for %s:\t %f"%(stat, t.timeit(number=1000)))
    except:
        t.print_exc()
```

Quickly Obsoleted Timeless Timings

Behold! Ordered by fastest to slowest snippets:

```
$ ./test_get.py
Time for for i in range(1000):
    for x in s:
        break:   0.249871
Time for for i in range(1000): next(iter(s)):    0.526266
Time for for i in range(1000): s.add(s.pop()):   0.658832
Time for for i in range(1000): list(s)[0]:   4.117106
Time for for i in range(1000): random.sample(s, 1):  21.851104
```

Faceplants for the Whole Family

Unsurprisingly, **manual iteration remains at least twice as fast** as the next fastest solution. Although the gap has decreased from the Bad Old Python 2.x days (in which manual iteration was at least four times as fast), it disappoints the PEP 20 zealot in me that the most verbose solution is the best. At least converting a set into a list just to extract the first element of the set is as horrible as expected. *Thank Guido, may his light continue to guide us.*

Surprisingly, the **RNG-based solution is absolutely horrible.** List conversion is bad, but `random` *really* takes the awful-sauce cake. So much for the Random Number God.

I just wish the amorphous They would PEP up a `set.get_first()` method for us already. If you're reading this, They: "Please. Do something."

edited Oct 15, 2016 at 6:40

answered Oct 15, 2016 at 3:00

Cecil Curry

10.2k ● 6 ● 42 ● 54

Share

Improve this answer

Follow

3 I think complaining that that `next(iter(s))` is twice slower than `for x in s: break` in `CPython` is kind of strange. I mean that is `CPython`. It will be about 50-100 times (or something like that) slower than C or Haskell doing the same thing (for the most of the time, especially so in iteration, no tail call elimination and no optimizations whatsoever.). Loosing some microseconds doesn't make a real difference. Don't you think? And there's also PyPy – user1685095 May 28, 2017 at 14:06

6 Since sets are not ordered, a `set.get_first()` could be misleading. But I would like a `set.get_any()`, which returns any element from the set, even if that element is always the same. – Eduardo Mar 19, 2022 at 2:06

In python 3.11, it seems s.add(s.pop()) works faster: ``` Time for for i in range(1000): for x in s: break: 0.044704 Time for for i in range(1000): next(iter(s)): 0.063221 Time for for i in range(1000): s.add(s.pop()): 0.056717 Time for for i in range(1000): list(s)[0]: 0.835464 ``` – Zhiyuan Chen Oct 19, 2023 at 13:51

40

To provide some timing figures behind the different approaches, consider the following code. *The get() is my custom addition to Python's setobject.c, being just a pop() without removing the element.*

```
from timeit import *

stats = ["for i in xrange(1000): iter(s).next()   ",
         "for i in xrange(1000): \n\tfor x in s: \n\t\tbreak",
         "for i in xrange(1000): s.add(s.pop())   ",
         "for i in xrange(1000): s.get()          "]

for stat in stats:
    t = Timer(stat, setup="s=set(range(100))")
    try:
        print "Time for %s:\t %f"%(stat, t.timeit(number=1000))
    except:
        t.print_exc()
```

The output is:

```
$ ./test_get.py
Time for for i in xrange(1000): iter(s).next()   :       0.433080
Time for for i in xrange(1000):
        for x in s:
                break:   0.148695
Time for for i in xrange(1000): s.add(s.pop())   :       0.317418
Time for for i in xrange(1000): s.get()          :       0.146673
```

This means that the ***for/break*** solution is the fastest (sometimes faster than the custom get() solution).

Share Improve this answer Follow

answered Oct 23, 2009 at 10:47

wr.

2,859 ● 1 ● 23 ● 28

Does anyone have an idea why iter(s).next() is so much slower than the other possibilities, even slower than s.add(s.pop())? For me it feels like very bad design of iter() and next() if the timings look like that. – peschü Jun 24, 2015 at 10:35

Well for one that line creates a new iter object each iteration. – Ryan Sep 13, 2015 at 19:37

5 @Ryan: Isn't an iterator object created implicitly for `for x in s` as well? "An iterator is created for the result of the `expression_list`." – musiphil Oct 5, 2015 at 17:25

2 @musiphil That is true; originally I missed the "break" one being at 0.14, that is really counter-intuitive. I want to do a deep dive into this when I have time. – Ryan Oct 5, 2015 at 17:36

1 I know this is old, but when adding `s.remove()` into the mix the `iter` examples both `for` and `iter` go catastrophically bad. – AChampion Jan 24, 2016 at 8:27

29

Since you want a random element, this will also work:

```
>>> import random
>>> s = set([1,2,3])
>>> random.sample(s, 1)
[2]
```

The documentation doesn't seem to mention performance of `random.sample`. From a really quick empirical test with a huge list and a huge set, it seems to be constant time for a list but not for the set. Also, iteration over a set isn't random; the order is undefined but predictable:

```
>>> list(set(range(10))) == range(10)
True
```

If randomness is important and you need a bunch of elements in constant time (large sets), I'd use `random.sample` and convert to a list first:

```
>>> lst = list(s) # once, O(len(s))?
...
>>> e = random.sample(lst, 1)[0] # constant time
```

Share Improve this answer

edited Sep 13, 2008 at 12:53

answered Sep 12, 2008 at 21:43

Follow

dF.
75.7k ● 31 ● 135 ● 137

14 If you just want one element, random.choice is more sensible. – Gregg Lind Nov 3, 2008 at 17:28

list(s).pop() will do if you don't care which element to take. – Evgeny May 12, 2014 at 0:11

11 @Gregg: You can't use `choice()`, because Python will try to index your set and that doesn't work. – Kevin Jan 20, 2015 at 21:13

5 While clever, this is actually **the slowest solution yet suggested by an order of magnitude.** Yes, it's *that* slow. Even converting the set into a list just to extract the first element of that list is faster. For the non-believers amongst us (*...hi!*), see these fabulous timings. – Cecil Curry Oct 15, 2016 at 3:03

27

Yet another way in Python 3:

```
next(iter(s))
```

or

```
s.__iter__().__next__()
```

Share

Improve this answer

Follow

edited May 27, 2021 at 14:48

answered Oct 14, 2020 at 12:04

dzang
2,260 ● 2 ● 14 ● 21

2 `next(iter(s))` will do the same thing, but will be shorter and more Pythonic. – Eerik Sven Puudist May 27, 2021 at 14:05

17

Seemingly the **most compact** (6 symbols) though **very slow** way to get a set element (made possible by PEP 3132):

```
e,*_=s
```

With Python 3.5+ you can also use this 7-symbol expression (thanks to PEP 448):

```
[*s][0]
```

Both options are roughly 1000 times slower on my machine than the for-loop method.

Share

Improve this answer

Follow

edited May 11, 2018 at 18:17

answered Aug 21, 2017 at 17:54

skovorodkin

10.2k ● 1 ● 40 ● 30

7 The for loop method (or more accurately the iterator method) has O(1) time complexity, while these methods are O(N). They are *concise* though. :) – ForeverWintr Sep 12, 2019 at 5:27

6

I use a utility function I wrote. Its name is somewhat misleading because it kind of implies it might be a random item or something like that.

```
def anyitem(iterable):
    try:
        return iter(iterable).next()
    except StopIteration:
        return None
```

Share Improve this answer Follow

answered Sep 14, 2008 at 4:57

Nick

22.2k ● 18 ● 50 ● 50

8 You can also go with next(iter(iterable), None) to save ink :) – 1'' Sep 21, 2017 at 5:20

6

Following @wr. post, I get similar results (for Python3.5)

```
from timeit import *

stats = ["for i in range(1000): next(iter(s))",
         "for i in range(1000): \n\tfor x in s: \n\t\tbreak",
         "for i in range(1000): s.add(s.pop())"]

for stat in stats:
    t = Timer(stat, setup="s=set(range(100000))")
    try:
        print("Time for %s:\t %f"%(stat, t.timeit(number=1000)))
    except:
        t.print_exc()
```

Output:

```
Time for for i in range(1000): next(iter(s)):    0.205888
Time for for i in range(1000):
    for x in s:
        break:                                   0.083397
Time for for i in range(1000): s.add(s.pop()):   0.226570
```

However, when changing the underlying set (e.g. call to `remove()`) things go badly for the iterable examples (`for`, `iter`):

```
from timeit import *

stats = ["while s:\n\ta = next(iter(s))\n\ts.remove(a)",
         "while s:\n\tfor x in s: break\n\ts.remove(x)",
         "while s:\n\tx=s.pop()\n\ts.add(x)\n\ts.remove(x)"]

for stat in stats:
    t = Timer(stat, setup="s=set(range(100000))")
    try:
        print("Time for %s:\t %f"%(stat, t.timeit(number=1000)))
    except:
        t.print_exc()
```

Results in:

```
Time for while s:
    a = next(iter(s))
    s.remove(a):              2.938494
Time for while s:
    for x in s: break
    s.remove(x):              2.728367
Time for while s:
    x=s.pop()
    s.add(x)
    s.remove(x):              0.030272
```

Share Improve this answer Follow

answered Jan 24, 2016 at 8:38

AChampion

30.2k ● 4 ● 62 ● 79

2

What I usually do for small collections is to create kind of parser/converter method like this

```
def convertSetToList(setName):
return list(setName)
```

Then I can use the new list and access by index number

```
userFields = convertSetToList(user)
name = request.json[userFields[0]]
```

As a list you will have all the other methods that you may need to work with

Share Improve this answer Follow

answered Mar 22, 2020 at 18:10

Josué Carvajal
112 ● 1 ● 9

11 why not just use `list` instead of creating a converter method? – Daren Thomas Mar 24, 2020 at 8:36

2

You can unpack the values to access the elements:

```
s = set([1, 2, 3])

v1, v2, v3 = s

print(v1,v2,v3)
#1 2 3
```

Share Improve this answer Follow

answered Oct 6, 2020 at 11:16

seralouk
33k ● 10 ● 123 ● 138

I suppose you could unpack to `v1, _*`. Without a wildcard, you'd need to exactly match the number of elements. But as noted in the previous answer stackoverflow.com/a/45803038/15416, this is slow – MSalters Aug 4, 2021 at 14:27

0

I f you want just the first element try this: b = (a-set()).pop()

Share Improve this answer Follow

answered Dec 23, 2020 at 19:39

Necho
484 ● 3 ● 5

1 Set is an unordered collection, so there's no such thing as "first element" :) – piit79 Sep 20, 2021 at 8:38

-4

How about `s.copy().pop()`? I haven't timed it, but it should work and it's simple. It works best for small sets however, as it copies the whole set.

Share Improve this answer Follow

answered Mar 6, 2018 at 19:14

Solomon Ucko
6,089 ● 3 ● 27 ● 48

-8

Another option is to use a dictionary with values you don't care about. E.g.,

```
poor_man_set = {}
poor_man_set[1] = None
poor_man_set[2] = None
poor_man_set[3] = None
...
```

You can treat the keys as a set except that they're just an array:

```
keys = poor_man_set.keys()
print "Some key = %s" % keys[0]
```

A side effect of this choice is that your code will be backwards compatible with older, pre-`set` versions of Python. It's maybe not the best answer but it's another option.

Edit: You can even do something like this to hide the fact that you used a dict instead of an array or set:

```
poor_man_set = {}
poor_man_set[1] = None
poor_man_set[2] = None
poor_man_set[3] = None
poor_man_set = poor_man_set.keys()
```

Share

Improve this answer

Follow

edited Sep 12, 2008 at 21:15

answered Sep 12, 2008 at 20:52

Pat Notz

214k ● 31 ● 94 ● 92

3 This doesn't work the way you hope it will. In python 2 keys() is an O(n) operation, so you're no longer constant time, but at least keys[0] will return the value you expect. In python 3 keys() is an O(1) operations, so yay! However, it no longer returns a list object, it returns a set-like object that can't be indexed, so keys[0] would throw TypeError. stackoverflow.com/questions/39219065/… – sage88 Apr 7, 2017 at 19:55