Usage of __slots__?

Asked 13 years, 10 months ago Modified 2 months ago Viewed 331k times

What is the purpose of `__slots__` in Python — especially with respect to when I would want to use it, and when not?

1130

python oop python-internals slots

Share Edit Follow

edited May 2, 2019 at 11:23

kmario23
53.9k 13 152 146

asked Jan 23, 2009 at 5:37

Jeb
15.4k 6 34 36

14 Answers

Sorted by: Highest score (default)

1628

In Python, what is the purpose of `__slots__` and what are the cases one should avoid this?

TLDR:

The special attribute `__slots__` allows you to explicitly state which instance attributes you expect your object instances to have, with the expected results:

1. **faster** attribute access.
2. **space savings** in memory.

The space savings is from

1. Storing value references in slots instead of `__dict__`.
2. Denying `__dict__` and `__weakref__` creation if parent classes deny them and you declare `__slots__`.

Quick Caveats

Small caveat, you should only declare a particular slot one time in an inheritance tree. For example:

```
class Base:
    __slots__ = 'foo', 'bar'

class Right(Base):
    __slots__ = 'baz',

class Wrong(Base):
    __slots__ = 'foo', 'bar', 'baz'        # redundant foo and bar
```

Python doesn't object when you get this wrong (it probably should), problems might not otherwise manifest, but your objects will take up more space than they otherwise should. Python 3.8:

```
>>> from sys import getsizeof
>>> getsizeof(Right()), getsizeof(Wrong())
(56, 72)
```

This is because the Base's slot descriptor has a slot separate from the Wrong's. This shouldn't usually come up, but it could:

```
>>> w = Wrong()
>>> w.foo = 'foo'
>>> Base.foo.__get__(w)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: foo
>>> Wrong.foo.__get__(w)
'foo'
```

The biggest caveat is for multiple inheritance - multiple "parent classes with nonempty slots" cannot be combined.

To accommodate this restriction, follow best practices: Factor out all but one or all parents' abstraction which their concrete class respectively and your new concrete class collectively will inherit from - giving the abstraction(s) empty slots (just like abstract base classes in the standard library).

See section on multiple inheritance below for an example.

Requirements:

- To have attributes named in `__slots__` to actually be stored in slots instead of a `__dict__`, a class must inherit from `object` (automatic in Python 3, but must be explicit in Python 2).
- To prevent the creation of a `__dict__`, you must inherit from `object` and all classes in the inheritance must declare `__slots__` and none of them can have a `'__dict__'` entry.

There are a lot of details if you wish to keep reading.

Why use `__slots__`: Faster attribute access.

The creator of Python, Guido van Rossum, states that he actually created `__slots__` for faster attribute access.

It is trivial to demonstrate measurably significant faster access:

```
import timeit

class Foo(object): __slots__ = 'foo',

class Bar(object): pass

slotted = Foo()
not_slotted = Bar()

def get_set_delete_fn(obj):
    def get_set_delete():
        obj.foo = 'foo'
        obj.foo
        del obj.foo
    return get_set_delete
```

and

```
>>> min(timeit.repeat(get_set_delete_fn(slotted)))
0.2846834529991611
>>> min(timeit.repeat(get_set_delete_fn(not_slotted)))
0.3664822799983085
```

The slotted access is almost 30% faster in Python 3.5 on Ubuntu.

```
>>> 0.3664822799983085 / 0.2846834529991611
1.2873325658284342
```

In Python 2 on Windows I have measured it about 15% faster.

Why use `__slots__`: Memory Savings

Another purpose of `__slots__` is to reduce the space in memory that each object instance takes up.

My own contribution to the documentation clearly states the reasons behind this:

> The space saved over using `__dict__` can be significant.

SQLAlchemy attributes a lot of memory savings to `__slots__`.

To verify this, using the Anaconda distribution of Python 2.7 on Ubuntu Linux, with `guppy.hpy` (aka heapy) and `sys.getsizeof`, the size of a class instance without `__slots__` declared, and nothing else, is 64 bytes. That does *not* include the `__dict__`. Thank you Python for lazy evaluation again, the `__dict__` is apparently not called into existence until it is referenced, but classes without data are usually useless. When called into existence, the `__dict__` attribute is a minimum of 280 bytes additionally.

In contrast, a class instance with `__slots__` declared to be `()` (no data) is only 16 bytes, and 56 total bytes with one item in slots, 64 with two.

For 64 bit Python, I illustrate the memory consumption in bytes in Python 2.7 and 3.6, for `__slots__` and `__dict__` (no slots defined) for each point where the dict grows in 3.6 (except for 0, 1, and 2 attributes):

```
       Python 2.7            Python 3.6
attrs  __slots__  __dict__*  __slots__  __dict__* | *(no slots defined)
none   16         56 + 272†  16         56 + 112† | †if __dict__ referenced
one    48         56 + 272   48         56 + 112
two    56         56 + 272   56         56 + 112
six    88         56 + 1040  88         56 + 152
11     128        56 + 1040  128        56 + 240
22     216        56 + 3344  216        56 + 408
43     384        56 + 3344  384        56 + 752
```

So, in spite of smaller dicts in Python 3, we see how nicely `__slots__` scale for instances to save us memory, and that is a major reason you would want to use `__slots__`.

Just for completeness of my notes, note that there is a one-time cost per slot in the class's namespace of 64 bytes in Python 2, and 72 bytes in Python 3, because slots use data descriptors like properties, called "members".

```
>>> Foo.foo
<member 'foo' of 'Foo' objects>
>>> type(Foo.foo)
<class 'member_descriptor'>
```

```
>>> getsizeof(Foo.foo)
72
```

Demonstration of `__slots__`:

To deny the creation of a `__dict__`, you must subclass `object`. Everything subclasses `object` in Python 3, but in Python 2 you had to be explicit:

```
class Base(object):
    __slots__ = ()
```

now:

```
>>> b = Base()
>>> b.a = 'a'
Traceback (most recent call last):
  File "<pyshell#38>", line 1, in <module>
    b.a = 'a'
AttributeError: 'Base' object has no attribute 'a'
```

Or subclass another class that defines `__slots__`

```
class Child(Base):
    __slots__ = ('a',)
```

and now:

```
c = Child()
c.a = 'a'
```

but:

```
>>> c.b = 'b'
Traceback (most recent call last):
  File "<pyshell#42>", line 1, in <module>
    c.b = 'b'
AttributeError: 'Child' object has no attribute 'b'
```

To allow `__dict__` creation while subclassing slotted objects, just add `'__dict__'` to the `__slots__` (note that slots are ordered, and you shouldn't repeat slots that are already in parent classes):

```
class SlottedWithDict(Child):
    __slots__ = ('__dict__', 'b')

swd = SlottedWithDict()
swd.a = 'a'
swd.b = 'b'
swd.c = 'c'
```

and

```
>>> swd.__dict__
{'c': 'c'}
```

Or you don't even need to declare `__slots__` in your subclass, and you will still use slots from the parents, but not restrict the creation of a `__dict__`:

```
class NoSlots(Child): pass
ns = NoSlots()
ns.a = 'a'
ns.b = 'b'
```

And:

```
>>> ns.__dict__
{'b': 'b'}
```

However, `__slots__` may cause problems for multiple inheritance:

```
class BaseA(object):
    __slots__ = ('a',)

class BaseB(object):
    __slots__ = ('b',)
```

Because creating a child class from parents with both non-empty slots fails:

```
>>> class Child(BaseA, BaseB): __slots__ = ()
Traceback (most recent call last):
  File "<pyshell#68>", line 1, in <module>
    class Child(BaseA, BaseB): __slots__ = ()
TypeError: Error when calling the metaclass bases
    multiple bases have instance lay-out conflict
```

If you run into this problem, You *could* just remove `__slots__` from the parents, or if you have control of the parents, give them empty slots, or refactor to abstractions:

```
from abc import ABC

class AbstractA(ABC):
    __slots__ = ()

class BaseA(AbstractA):
    __slots__ = ('a',)

class AbstractB(ABC):
    __slots__ = ()

class BaseB(AbstractB):
    __slots__ = ('b',)

class Child(AbstractA, AbstractB):
    __slots__ = ('a', 'b')

c = Child() # no problem!
```

Add `'__dict__'` to `__slots__` to get dynamic assignment:

```
class Foo(object):
    __slots__ = 'bar', 'baz', '__dict__'
```

and now:

```
>>> foo = Foo()
>>> foo.boink = 'boink'
```

So with `'__dict__'` in slots we lose some of the size benefits with the upside of having dynamic assignment and still having slots for the names we do expect.

When you inherit from an object that isn't slotted, you get the same sort of semantics when you use `__slots__` - names that are in `__slots__` point to slotted values, while any other values are put in the instance's `__dict__`.

Avoiding `__slots__` because you want to be able to add attributes on the fly is actually not a good reason - just add `"__dict__"` to your `__slots__` if this is required.

You can similarly add `__weakref__` to `__slots__` explicitly if you need that feature.

Set to empty tuple when subclassing a namedtuple:

The namedtuple builtin make immutable instances that are very lightweight (essentially, the size of tuples) but to get the benefits, you need to do it yourself if you subclass them:

```
from collections import namedtuple
class MyNT(namedtuple('MyNT', 'bar baz')):
    """MyNT is an immutable and lightweight object"""
    __slots__ = ()
```

usage:

```
>>> nt = MyNT('bar', 'baz')
>>> nt.bar
'bar'
>>> nt.baz
'baz'
```

And trying to assign an unexpected attribute raises an `AttributeError` because we have prevented the creation of `__dict__`:

```
>>> nt.quux = 'quux'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'MyNT' object has no attribute 'quux'
```

You *can* allow `__dict__` creation by leaving off `__slots__ = ()`, but you can't use non-empty `__slots__` with subtypes of tuple.

Biggest Caveat: Multiple inheritance

Even when non-empty slots are the same for multiple parents, they cannot be used together:

```
class Foo(object):
    __slots__ = 'foo', 'bar'
class Bar(object):
    __slots__ = 'foo', 'bar' # alas, would work if empty, i.e. ()

>>> class Baz(Foo, Bar): pass
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
TypeError: Error when calling the metaclass bases
    multiple bases have instance lay-out conflict
```

Using an empty `__slots__` in the parent seems to provide the most flexibility, **allowing the child to choose to prevent or allow** (by adding `'__dict__'` to get dynamic assignment, see section above) **the creation of a `__dict__`** :

```
class Foo(object): __slots__ = ()
class Bar(object): __slots__ = ()
class Baz(Foo, Bar): __slots__ = ('foo', 'bar')
b = Baz()
b.foo, b.bar = 'foo', 'bar'
```

You don't *have* to have slots - so if you add them, and remove them later, it shouldn't cause any problems.

Going out on a limb here: If you're composing mixins or using abstract base classes, which aren't intended to be instantiated, an empty `__slots__` in those parents seems to be the best way to go in terms of flexibility for subclassers.

To demonstrate, first, let's create a class with code we'd like to use under multiple inheritance

```
class AbstractBase:
    __slots__ = ()
    def __init__(self, a, b):
        self.a = a
        self.b = b
    def __repr__(self):
        return f'{type(self).__name__}({repr(self.a)}, {repr(self.b)})'
```

We could use the above directly by inheriting and declaring the expected slots:

```
class Foo(AbstractBase):
    __slots__ = 'a', 'b'
```

But we don't care about that, that's trivial single inheritance, we need another class we might also inherit from, maybe with a noisy attribute:

```
class AbstractBaseC:
    __slots__ = ()
    @property
    def c(self):
        print('getting c!')
        return self._c
    @c.setter
    def c(self, arg):
        print('setting c!')
        self._c = arg
```

Now if both bases had nonempty slots, we couldn't do the below. (In fact, if we wanted, we could have given `AbstractBase` nonempty slots a and b, and left them out of the below declaration - leaving them in would be wrong):

```
class Concretion(AbstractBase, AbstractBaseC):
    __slots__ = 'a b _c'.split()
```

And now we have functionality from both via multiple inheritance, and can still deny `__dict__` and `__weakref__` instantiation:

```
>>> c = Concretion('a', 'b')
>>> c.c = c
setting c!
```

```
>>> c.c
getting c!
Concretion('a', 'b')
>>> c.d = 'd'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Concretion' object has no attribute 'd'
```

Other cases to avoid slots:

- Avoid them when you want to perform `__class__` assignment with another class that doesn't have them (and you can't add them) unless the slot layouts are identical. (I am very interested in learning who is doing this and why.)
- Avoid them if you want to subclass variable length builtins like long, tuple, or str, and you want to add attributes to them.
- Avoid them if you insist on providing default values via class attributes for instance variables.

You may be able to tease out further caveats from the rest of the `__slots__` documentation (the 3.7 dev docs are the most current), which I have made significant recent contributions to.

Critiques of other answers

The current top answers cite outdated information and are quite hand-wavy and miss the mark in some important ways.

Do not "only use `__slots__` when instantiating lots of objects"

I quote:

> "You would want to use `__slots__` if you are going to instantiate a lot (hundreds, thousands) of objects of the same class."

Abstract Base Classes, for example, from the `collections` module, are not instantiated, yet `__slots__` are declared for them.

Why?

If a user wishes to deny `__dict__` or `__weakref__` creation, those things must not be available in the parent classes.

`__slots__` contributes to reusability when creating interfaces or mixins.

It is true that many Python users aren't writing for reusability, but when you are, having the option to deny unnecessary space usage is valuable.

`__slots__` doesn't break pickling

When pickling a slotted object, you may find it complains with a misleading `TypeError`:

```
>>> pickle.loads(pickle.dumps(f))
TypeError: a class that defines __slots__ without defining __getstate__ cannot
be pickled
```

This is actually incorrect. This message comes from the oldest protocol, which is the default. You can select the latest protocol with the `-1` argument. In Python 2.7 this would be `2` (which was introduced in 2.3), and in 3.6 it is `4`.

```
>>> pickle.loads(pickle.dumps(f, -1))
```

```
<__main__.Foo object at 0x1129C770>
```

in Python 2.7:

```
>>> pickle.loads(pickle.dumps(f, 2))
<__main__.Foo object at 0x1129C770>
```

in Python 3.6

```
>>> pickle.loads(pickle.dumps(f, 4))
<__main__.Foo object at 0x1129C770>
```

So I would keep this in mind, as it is a solved problem.

Critique of the (until Oct 2, 2016) accepted answer

The first paragraph is half short explanation, half predictive. Here's the only part that actually answers the question

> The proper use of `__slots__` is to save space in objects. Instead of having a dynamic dict that allows adding attributes to objects at anytime, there is a static structure which does not allow additions after creation. This saves the overhead of one dict for every object that uses slots

The second half is wishful thinking, and off the mark:

> While this is sometimes a useful optimization, it would be completely unnecessary if the Python interpreter was dynamic enough so that it would only require the dict when there actually were additions to the object.

Python actually does something similar to this, only creating the `__dict__` when it is accessed, but creating lots of objects with no data is fairly ridiculous.

The second paragraph oversimplifies and misses actual reasons to avoid `__slots__`. The below is *not* a real reason to avoid slots (for *actual* reasons, see the rest of my answer above.):

> They change the behavior of the objects that have slots in a way that can be abused by control freaks and static typing weenies.

It then goes on to discuss other ways of accomplishing that perverse goal with Python, not discussing anything to do with `__slots__`.

The third paragraph is more wishful thinking. Together it is mostly off-the-mark content that the answerer didn't even author and contributes to ammunition for critics of the site.

Memory usage evidence

Create some normal objects and slotted objects:

```
>>> class Foo(object): pass
>>> class Bar(object): __slots__ = ()
```

Instantiate a million of them:

```
>>> foos = [Foo() for f in xrange(1000000)]
>>> bars = [Bar() for b in xrange(1000000)]
```

Inspect with `guppy.hpy().heap()`:

```
>>> guppy.hpy().heap()
Partition of a set of 2028259 objects. Total size = 99763360 bytes.
 Index  Count   %     Size   % Cumulative  % Kind (class / dict of class)
     0 1000000  49 64000000  64  64000000  64 __main__.Foo
     1     169   0 16281480  16  80281480  80 list
     2 1000000  49 16000000  16  96281480  97 __main__.Bar
     3   12284   1   987472   1  97268952  97 str
...
```

Access the regular objects and their `__dict__` and inspect again:

```
>>> for f in foos:
...     f.__dict__
>>> guppy.hpy().heap()
Partition of a set of 3028258 objects. Total size = 379763480 bytes.
 Index  Count   %      Size    % Cumulative  % Kind (class / dict of class)
     0 1000000  33 280000000  74 280000000  74 dict of __main__.Foo
     1 1000000  33  64000000  17 344000000  91 __main__.Foo
     2     169   0  16281480   4 360281480  95 list
     3 1000000  33  16000000   4 376281480  99 __main__.Bar
     4   12284   0    987472   0 377268952  99 str
...
```

This is consistent with the history of Python, from Unifying types and classes in Python 2.2

> If you subclass a built-in type, extra space is automatically added to the instances to accomodate `__dict__` and `__weakrefs__`. (The `__dict__` is not initialized until you use it though, so you shouldn't worry about the space occupied by an empty dictionary for each instance you create.) If you don't need this extra space, you can add the phrase "`__slots__ = []`" to your class.

Share Edit Follow

edited Jun 30, 2021 at 18:07

answered Jan 21, 2015 at 4:46

Russia Must Remove Putin ♦

356k 85 395 329

151 This answer should be part of the official Python documentation about `__slots__`. Seriously! Thank you! – NightElfik May 10, 2018 at 23:53

54 @NightElfik believe it or not, I contributed to the Python docs on `__slots__` about a year back: github.com/python/cpython/pull/1819/files – Russia Must Remove Putin ♦ Jun 28, 2018 at 17:15

4 Fantastically detailed answer. I have one question: should one being using slots as default *unless* the usage hits one of the caveats, or are slots something to consider if you know you are going to struggle for speed/memory? To put it another way, should you encourage a newbie to learn about them and use them from the start? – freethebees Jul 11, 2019 at 9:59

5 @pepoluan no you do not need to list method names in `__slots__` - but thanks for the question! A slot declaration creates a descriptor object in the namespace (the `__dict__`) like a method definition does. – Russia Must Remove Putin ♦ Sep 28, 2020 at 13:43

3 @greatvovan thanks for bringing that to my attention, I have updated the text in two locations to make that point explicit. Let me know if that's good or if you think I've missed any other spots or any other issues as well. Much appreciated. – Russia Must Remove Putin ♦ Jun 30, 2021 at 18:08

Quoting Jacob Hallen:

277

The proper use of `__slots__` is to save space in objects. Instead of having a dynamic dict that allows adding attributes to objects at anytime, there is a static structure which does not allow additions after creation. [This use of `__slots__` eliminates the overhead of one dict for every object.] While this is sometimes a useful optimization, it would be completely unnecessary if the Python interpreter was dynamic enough so that it would only require the dict when there actually were additions to the object.

Unfortunately there is a side effect to slots. They change the behavior of the objects that have slots in a way that can be abused by control freaks and static typing weenies. This is bad, because the control freaks should be abusing the metaclasses and the static typing weenies should be abusing decorators, since in Python, there should be only one obvious way of doing something.

Making CPython smart enough to handle saving space without `__slots__` is a major undertaking, which is probably why it is not on the list of changes for P3k (yet).

Share Edit Follow

edited Nov 9, 2016 at 19:36

Rob Bednark
24.2k 21 78 120

answered Jan 23, 2009 at 5:54

Jeff Bauer
13.7k 9 51 73

94 I'd like to see some elaboration on the "static typing"/decorator point, sans pejoratives. Quoting absent third parties is unhelpful. `__slots__` doesn't address the same issues as static typing. For example, in C++, it is not the declaration of a member variable is being restricted, it is the assignment of an unintended type (and compiler enforced) to that variable. I'm not condoning the use of `__slots__` , just interested in the conversation. Thanks! – hiwaylon Nov 28, 2011 at 17:54

"in Python, there should be only one obvious way of doing something" So what is the one obvious way of preventing global variables (uppercase variables being named consts) using metaclasses? – dbow Feb 15 at 8:21

142

You would want to use `__slots__` if you are going to instantiate a lot (hundreds, thousands) of objects of the same class. `__slots__` only exists as a memory optimization tool.

It's highly discouraged to use `__slots__` for constraining attribute creation.

Pickling objects with `__slots__` won't work with the default (oldest) pickle protocol; it's necessary to specify a later version.

Some other introspection features of python may also be adversely affected.

Share Edit Follow

edited Oct 24, 2019 at 12:02

Toby Speight
25.9k 47 63 97

answered Jan 23, 2009 at 5:50

Ryan
14.8k 6 48 50

15 I demonstrate pickling a slotted object in my answer and also address the first part of your answer. – Russia Must Remove Putin ♦ May 2, 2016 at 15:08

7 I see your point, but slots offer faster attribute access as well (as others have stated). In that case you don't need *"to instantiate a lot (hundreds, thousands) of objects of the same class"* in order to gain performance. What you need instead are *a lot* of accesses to the same (slotted) attribute of the same instance. (Please correct me if I'm wrong.) – Rotareti Jul 19, 2017 at 4:28

6 why is it "highly discouraged"? I was recently looking for a way to constrain dynamic attribute creation. I found something but there was no mention of slots. Now I read about slots and it seems like exactly what i was looking for before. Whats wrong about using slots to prevent adding attributes at runtime?
– 463035818_is_not_a_number Jun 17, 2020 at 9:08

1 @idclev463035818 I don't think there is any wrong thing about that. – Ekrem Dinçel Oct 30, 2020 at 19:22

74

Each python object has a `__dict__` atttribute which is a dictionary containing all other attributes. e.g. when you type `self.attr` python is actually doing `self.__dict__['attr']` . As you can imagine using a dictionary to store attribute takes some extra space & time for accessing it.

However, when you use `__slots__`, any object created for that class won't have a `__dict__` attribute. Instead, all attribute access is done directly via pointers.

So if want a C style structure rather than a full fledged class you can use `__slots__` for compacting size of the objects & reducing attribute access time. A good example is a Point class containing attributes x & y. If you are going to have a lot of points, you can try using `__slots__` in order to conserve some memory.

Share Edit Follow

edited May 5, 2011 at 8:39

muhuk
15.5k 9 58 98

answered Jan 23, 2009 at 13:38

Suraj
4,719 1 18 12

11 No, an instance of a class with `__slots__` defined is *not* like a C-style structure. There is a class-level dictionary mapping attribute names to indexes, otherwise the following would not be possible: `class A(object): __slots__= "value",\n\na=A(); setattr(a, 'value', 1)` I really think this answer should be clarified (I can do that if you want). Also, I'm not certain that `instance.__hidden_attributes[instance.__class__[attrname]]` is faster than `instance.__dict__[attrname]` . – tzot Oct 15, 2011 at 13:56

30

In addition to the other answers, here is an example of using `__slots__`:

```
>>> class Test(object):   #Must be new-style class!
...  __slots__ = ['x', 'y']
...
>>> pt = Test()
>>> dir(pt)
['__class__', '__delattr__', '__doc__', '__getattribute__', '__hash__',
 '__init__', '__module__', '__new__', '__reduce__', '__reduce_ex__',
 '__repr__', '__setattr__', '__slots__', '__str__', 'x', 'y']
>>> pt.x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: x
>>> pt.x = 1
>>> pt.x
1
>>> pt.z = 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Test' object has no attribute 'z'
>>> pt.__dict__
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Test' object has no attribute '__dict__'
>>> pt.__slots__
['x', 'y']
```

So, to implement `__slots__`, it only takes an extra line (and making your class a new-style class if it isn't already). This way you can reduce the memory footprint of those classes 5-fold, at the expense of having to write custom pickle code, if and when that becomes necessary.

Share Edit Follow

answered Jun 3, 2015 at 7:43

Evgeni Sergeev
21.8k 16 104 123

14

Slots are very useful for library calls to eliminate the "named method dispatch" when making function calls. This is mentioned in the SWIG documentation. For high performance libraries that want to reduce function overhead for commonly called functions using slots is much faster.

Now this may not be directly related to the OPs question. It is related more to building extensions than it does to using the **slots** syntax on an object. But it does help complete the picture for the usage of slots and some of the reasoning behind them.

Share Edit Follow

answered Nov 25, 2012 at 3:06

Demolishun

1,592 12 15

A very simple example of `__slot__` attribute.

Problem: Without `__slots__`

If I don't have `__slot__` attribute in my class, I can add new attributes to my objects.

```
class Test:
    pass

obj1=Test()
obj2=Test()

print(obj1.__dict__)  #--> {}
obj1.x=12
print(obj1.__dict__)  # --> {'x': 12}
obj1.y=20
print(obj1.__dict__)  # --> {'x': 12, 'y': 20}

obj2.x=99
print(obj2.__dict__)  # --> {'x': 99}
```

If you look at example above, you can see that **obj1** and **obj2** have their own **x** and **y** attributes and python has also created a `dict` attribute for each object (**obj1** and **obj2**).

Suppose if my class **Test** has thousands of such objects? Creating an additional attribute `dict` for each object will cause lot of overhead (memory, computing power etc.) in my code.

Solution: With `__slots__`

Now in the following example my class **Test** contains `__slots__` attribute. Now I can't add new attributes to my objects (except attribute `x`) and python doesn't create a `dict` attribute anymore. This eliminates overhead for each object, which can become significant if you have many objects.

```
class Test:
    __slots__=("x")

obj1=Test()
obj2=Test()
obj1.x=12
print(obj1.x)  # --> 12
obj2.x=99
print(obj2.x)  # --> 99

obj1.y=28
print(obj1.y)  # --> AttributeError: 'Test' object has no attribute 'y'
```

Share Edit Follow

edited Jun 20, 2020 at 9:12
Community Bot
1 1

answered Nov 2, 2016 at 9:18
N Randhawa
8,253 3 42 47

Hi, you can use `class Test: __slots__= ("x", "__dict__")` and it will work. – Jan May 13 at 9:34

11

An attribute of a class instance has 3 properties: the instance, the name of the attribute, and the value of the attribute.

In ***regular attribute access***, the instance acts as a dictionary and the name of the attribute acts as the key in that dictionary looking up value.

instance(attribute) --> value

In ***__slots__ access***, the name of the attribute acts as the dictionary and the instance acts as the key in the dictionary looking up value.

attribute(instance) --> value

In ***flyweight pattern***, the name of the attribute acts as the dictionary and the value acts as the key in that dictionary looking up the instance.

attribute(value) --> instance

Share Edit Follow

answered Jun 4, 2014 at 22:09

Dmitry Rubanovich
2,441 18 27

1 This is a good share, and won't fit well in a comment on one of the answers that also suggest flyweights, but it is not a complete answer to the question itself. In particular (in just context of the question): why Flyweight, and "what are the cases one should avoid ..." `__slots__` ? – Merlyn Morgan-Graham Jul 25, 2014 at 6:22

@Merlyn Morgan-Graham, it serves as a hint on which to pick: regular access, __slots__, or flyweight. – Dmitry Rubanovich Jul 26, 2014 at 23:04

6

In addition to the other answers, `__slots__` also adds a little typographical security by limiting attributes to a predefined list. This has long been a problem with JavaScript which also allows you to add new attributes to an existing object, whether you meant to or not.

Here is a normal unslotted object which does nothing, but allows you to add attributes:

```
class Unslotted:
    pass
test = Unslotted()
test.name = 'Fred'
test.Name = 'Wilma'
```

Since Python is case sensitive, the two attributes, spelled the same but with different case, are different. If you suspect that one of those is a typing error, then bad luck.

Using slots, you can limit this:

```
class Slotted:
    __slots__ = ('name')
    pass
test = Slotted()
test.name = 'Fred'      #   OK
test.Name = 'Wilma'     #   Error
```

This time, the second attribute (`Name`) is disallowed because it's not in the `__slots__` collection.

I would suggest that it's probably better to use `__slots__` where possible to keep more control over the object.

Share Edit Follow

edited May 20 at 0:55

answered May 18 at 1:33

Manngo
12.4k 9 74 94

Beginning in Python 3.9, a `dict` may be used to add descriptions to attributes via `__slots__`. `None` may be used for attributes without descriptions, and private variables will not appear even if a description is given.

5

```
class Person:

    __slots__ = {
        "birthday":
            "A datetime.date object representing the person's birthday.",
        "name":
            "The first and last name.",
        "public_variable":
            None,
        "_private_variable":
            "Description",
    }


help(Person)
"""
Help on class Person in module __main__:

class Person(builtins.object)
 |  Data descriptors defined here:
 |
 |  birthday
 |      A datetime.date object representing the person's birthday.
 |
 |  name
 |      The first and last name.
 |
 |  public_variable
"""
```

Share Edit Follow

answered Apr 3 at 21:56

Simply Beautiful Art

924 12 13

> Was just looking if someone mentioned it *If a dictionary is used to assign **slots**, the dictionary keys will be used as the slot names. The values of the dictionary can be used to provide per-attribute docstrings that will be recognised by inspect.getdoc() and displayed in the output of help().* – Thingamabobs Jul 3 at 5:04

3

Another somewhat obscure use of `__slots__` is to add attributes to an object proxy from the ProxyTypes package, formerly part of the PEAK project. Its `ObjectWrapper` allows you to proxy another object, but intercept all interactions with the proxied object. It is not very commonly used (and no Python 3 support), but we have used it to implement a thread-safe blocking wrapper around an async implementation based on tornado that bounces all access to the proxied object through the ioloop, using thread-safe `concurrent.Future` objects to synchronise and return results.

By default any attribute access to the proxy object will give you the result from the proxied object. If you need to add an attribute on the proxy object, `__slots__` can be used.

```
from peak.util.proxies import ObjectWrapper

class Original(object):
    def __init__(self):
        self.name = 'The Original'

class ProxyOriginal(ObjectWrapper):

    __slots__ = ['proxy_name']

    def __init__(self, subject, proxy_name):
        # proxy_info attributed added directly to the
        # Original instance, not the ProxyOriginal instance
        self.proxy_info = 'You are proxied by {}'.format(proxy_name)

        # proxy_name added to ProxyOriginal instance, since it is
```

```
        # defined in __slots__
        self.proxy_name = proxy_name

        super(ProxyOriginal, self).__init__(subject)

if __name__ == "__main__":
    original = Original()
    proxy = ProxyOriginal(original, 'Proxy Overlord')

    # Both statements print "The Original"
    print "original.name: ", original.name
    print "proxy.name: ", proxy.name

    # Both statements below print
    # "You are proxied by Proxy Overlord", since the ProxyOriginal
    # __init__ sets it to the original object
    print "original.proxy_info: ", original.proxy_info
    print "proxy.proxy_info: ", proxy.proxy_info

    # prints "Proxy Overlord"
    print "proxy.proxy_name: ", proxy.proxy_name
    # Raises AttributeError since proxy_name is only set on
    # the proxy object
    print "original.proxy_name: ", proxy.proxy_name
```

Share Edit Follow

answered Feb 7, 2017 at 20:56

NeilenMarais
2,841 1 24 22

1

In addition to the myriad advantages described in other answers herein – compact instances for the memory-conscious, less error-prone than the more mutable `__dict__` -bearing instances, *et cetera* – I find that using `__slots__` offers more legible class declarations, as the instance variables of the class are explicitly out in the open.

To contend with inheritance issues with `__slots__` declarations I use this metaclass:

```
import abc

class Slotted(abc.ABCMeta):

    """ A metaclass that ensures its classes, and all subclasses,
        will be slotted types.
    """

    def __new__(metacls, name, bases, attributes, **kwargs):
        """ Override for `abc.ABCMeta.__new__(…)` setting up a
            derived slotted class.
        """
        if '__slots__' not in attributes:
            attributes['__slots__'] = tuple()

        return super(Slotted, metacls).__new__(metacls, name, # type: ignore
                                               bases,
                                               attributes,
                                              **kwargs)
```

… which, if declared as the metaclass of the base class in an inheritance tower, ensures that everything that derives from that base class will properly inherit `__slots__` attributes, even if an intermediate class fails to declare any. Like so:

```
# note no __slots__ declaration necessary with the metaclass:
class Base(metaclass=Slotted):
    pass
```

```
# class is properly slotted, no __dict__:
class Derived(Base):
    __slots__ = 'slot', 'another_slot'

# class is also properly slotted:
class FurtherDerived(Derived):
    pass
```

Share Edit Follow

answered Sep 11 at 6:46

fish2000
4,229 2 39 73

1

The original question was about general use cases not only about memory. So it should be mentioned here that you also get better *performance* when instantiating large amounts of objects - interesting e.g. when parsing large documents into objects or from a database.

Here is a comparison of creating object trees with a million entries, using slots and without slots. As a reference also the performance when using plain dicts for the trees (Py2.7.10 on OSX):

```
********** RUN 1 **********
1.96036410332 <class 'css_tree_select.element.Element'>
3.02922606468 <class 'css_tree_select.element.ElementNoSlots'>
2.90828204155 dict
********** RUN 2 **********
1.77050495148 <class 'css_tree_select.element.Element'>
3.10655999184 <class 'css_tree_select.element.ElementNoSlots'>
2.84120798111 dict
********** RUN 3 **********
1.84069895744 <class 'css_tree_select.element.Element'>
3.21540498734 <class 'css_tree_select.element.ElementNoSlots'>
2.59615707397 dict
********** RUN 4 **********
1.75041103363 <class 'css_tree_select.element.Element'>
3.17366290092 <class 'css_tree_select.element.ElementNoSlots'>
2.70941114426 dict
```

Test classes (ident, appart from slots):

```
class Element(object):
    __slots__ = ['_typ', 'id', 'parent', 'childs']
    def __init__(self, typ, id, parent=None):
        self._typ = typ
        self.id = id
        self.childs = []
        if parent:
            self.parent = parent
            parent.childs.append(self)

class ElementNoSlots(object): (same, w/o slots)
```

testcode, verbose mode:

```
na, nb, nc = 100, 100, 100
for i in (1, 2, 3, 4):
    print '*' * 10, 'RUN', i, '*' * 10
    # tree with slot and no slot:
    for cls in Element, ElementNoSlots:
        t1 = time.time()
        root = cls('root', 'root')
        for i in xrange(na):
            ela = cls(typ='a', id=i, parent=root)
```

```
        for j in xrange(nb):
            elb = cls(typ='b', id=(i, j), parent=ela)
            for k in xrange(nc):
                elc = cls(typ='c', id=(i, j, k), parent=elb)
    to =  time.time() - t1
    print to, cls
    del root

# ref: tree with dicts only:
t1 = time.time()
droot = {'childs': []}
for i in xrange(na):
    ela =  {'typ': 'a', id: i, 'childs': []}
    droot['childs'].append(ela)
    for j in xrange(nb):
        elb =  {'typ': 'b', id: (i, j), 'childs': []}
        ela['childs'].append(elb)
        for k in xrange(nc):
            elc =  {'typ': 'c', id: (i, j, k), 'childs': []}
            elb['childs'].append(elc)
td = time.time() - t1
print td, 'dict'
del droot
```

Share Edit Follow

answered Jan 12, 2016 at 18:42

Red Pill
471 5 15

1

You have — essentially — no use for `__slots__` .

For the time when you think you might need `__slots__` , you actually want to use **Lightweight** or **Flyweight** design patterns. These are cases when you no longer want to use purely Python objects. Instead, you want a Python object-like wrapper around an array, struct, or numpy array.

```
class Flyweight(object):

    def get(self, theData, index):
        return theData[index]

    def set(self, theData, index, value):
        theData[index]= value
```

The class-like wrapper has no attributes — it just provides methods that act on the underlying data. The methods can be reduced to class methods. Indeed, it could be reduced to just functions operating on the underlying array of data.

Share Edit Follow

edited Aug 24, 2016 at 14:11

Noctis Skytower
20.9k 16 81 114

answered Jan 23, 2009 at 11:15

S.Lott
378k 79 503 773

18 What has Flyweight to do with `__slots__` ? – oefe Jan 24, 2009 at 22:46

3 @oefe: I certainly don't get your question. I can quote my answer, if it helps "when you think you might need **slots**, you actually want to use ... Flyweight design pattern". That's what Flyweight has to do with **slots**. Do you have a more specific question? – S.Lott Jan 24, 2009 at 23:41

21 @oefe: Flyweight and `__slots__` are both optimization techniques to save memory. `__slots__` shows benefits when you have many many objects as well as Flyweight design pattern. The both solve the same problem. – jfs Nov 29, 2009 at 20:51

7 Is there a available comparison between using slots and using Flyweight regarding memory consumption and speed? – kontulai Apr 23, 2013 at 4:11

9 Although Flyweight is certainly useful in some contexts, believe it or not, the answer to "how can I reduce memory usage in Python when I create a zillion objects" is not always "don't use Python for your zillion objects." Sometimes `__slots__` really is the answer, and as Evgeni points out, it can be added as a simple afterthought (e.g.

you can focus on correctness first, and then add performance). – Patrick Maupin Jul 25, 2015 at 16:19

Highly active question. Earn 10 reputation (not counting the association bonus) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.