Python class inherits object

Ask Question

Is there any reason for a class declaration to inherit from `object` ?

I just found some code that does this and I can't find a good reason why.

```
class MyClass(object):
    # class code follows...
```

python class object inheritance

edited Jan 15 '13 at 11:08
Jerub
30.1k 11 58 87

asked Oct 25 '10 at 14:15
tjvr
3,877 5 16 23

38 The answer to this question (while simple) is quite difficult to find. Googling things like "python object base class" or similar comes up with pages and pages of tutorials on object oriented programming. Upvoting because this is the first link that led me to the search terms "old vs. new-style python objects" – vastlysuperiorman Dec 22 '15 at 20:42

7 Answers

> **Is there any reason for a class declaration to inherit from `object` ?**

tl;dr: In Python 3, apart from compatibility between Python 2 and 3, *no reason*. In Python 2, *many reasons*.

Python 2.x story:

In Python 2.x (from 2.2 onwards) there's two styles of classes depending on the presence or absence of `object` as a base-class:

1. **"classic" style** classes: they don't have `object` as a base class:

```
>>> class ClassicSpam:      # no base class
...     pass
>>> ClassicSpam.__bases__
()
```

2. **"new" style** classes: they have, directly *or indirectly* (e.g inherit from a built-in type), `object` as a base class:

```
>>> class NewSpam(object):           # directly inherit from object
...    pass
>>> NewSpam.__bases__
(<type 'object'>,)
>>> class IntSpam(int):              # indirectly inherit from object...
...    pass
>>> IntSpam.__bases__
(<type 'int'>,)
>>> IntSpam.__bases__[0].__bases__   # ... because int inherits from object
(<type 'object'>,)
```

Without a doubt, when writing a class you'll *always* want to go for new-style classes. The perks of doing so are numerous, to list some of them:

- Support for descriptors. Specifically, the following constructs are made possible with descriptors:
    1. `classmethod` : A method that receives the class as an implicit argument instead of the instance.
    2. `staticmethod` : A method that does not receive the implicit argument `self` as a first argument.
    3. properties with `property` : Create functions for managing the getting, setting and deleting of an attribute.
    4. `__slots__` : Saves memory consumptions of a class and also results in faster attribute access. Of course, it does impose limitations.
- The `__new__` static method: lets you customize how new class instances are created.

- Method resolution order (MRO): in what order the base classes of a class will be searched when trying to resolve which method to call.
- Related to MRO, `super` calls. Also see, `super()` considered super.

If you don't inherit from `object`, forget these. A more exhaustive description of the previous bullet points along with other perks of "new" style classes can be found here.

One of the downsides of new-style classes is that the class itself is more memory demanding. Unless you're creating many class objects, though, I doubt this would be an issue and it's a negative sinking in a sea of positives.

Python 3.x story:

In Python 3, things are simplified. Only new-style classes exist (referred to plainly as classes) so, the only difference in adding `object` is requiring you to type in 8 more characters. This:

```
class ClassicSpam:
    pass
```

is completely equivalent (apart from their name :-) to this:

```
class NewSpam(object):
     pass
```

and to thi

```
clas
    pass
```

all have `object` in their `__bases__`.

```
>>> [object in cls.__bases__ for cls in {Spam, NewSpam, ClassicSpam}]
[True, True, True]
```

So, what should you do?

In Python 2: *always inherit from `object` explicitly*. Get the perks.

In Python 3: inherit from `object` if you are writing code that tries to be Python agnostic, that is, it needs to work both in Python 2 and in Python 3. Otherwise don't, it really makes no difference since Python inserts it for you behind the scenes.

edited Mar 21 at 12:24

answered Jul 12 '17 at 15:34
Jim Fasarakis Hilliard
62.1k 14 129 159

"depending on the presence or absence of a built-in type as a base-class" => actually it's not about "absence or presence of a builtin type as a base class" but wether the class inherits - directly **or indirectly** - from `object`. IIRC there was a point in time where not all builtin types where ported to new-style classes yet. – bruno desthuilliers Mar 19 at 15:31

@brunodesthuilliers My impression was that all built-in types did inherit from `object`. I do have a Python 2.2.3 around and after a quick check I couldn't find an offender but, I'll reword the answer later to make it more clear. Would be interested if you could find an example though, my curiosity is piqued. – Jim Fasarakis Hilliard Mar 20 at 9:50

In all honesty (cf the "IIRC" in my previous comment) I am not 101% sure about this point (wether all builtin types were already converted to new-style classes when new-style classes were introduced) - I might just be plain wrong, or this might only have concerned some of the standard lib's (but not builtin) types. But yet I think it should be better to clarify that what makes a new-style class is having `object` in it's bases. – bruno desthuilliers Mar 20 at 11:18

Python 3.x:
`class MyClass(object):` = new-style class
`class MyClass:` = new-style class (implicitly inherits from object)

Python 2.x:
`class MyClass(object):` = new-style class
`class MyClass:` = *OLD-STYLE CLASS*

Explanation: