Creating a singleton in Python

Asked 11 years, 4 months ago Modified 1 month ago Viewed 597k times

1514

This question is not for the discussion of whether or not the singleton design pattern is desirable, is an anti-pattern, or for any religious wars, but to discuss how this pattern is best implemented in Python in such a way that is most pythonic. In this instance I define 'most pythonic' to mean that it follows the 'principle of least astonishment'.

I have multiple classes which would become singletons (my use-case is for a logger, but this is not important). I do not wish to clutter several classes with added gumph when I can simply inherit or decorate.

Best methods:

Method 1: A decorator

```
def singleton(class_):
    instances = {}
    def getinstance(*args, **kwargs):
        if class_ not in instances:
            instances[class_] = class_(*args, **kwargs)
        return instances[class_]
    return getinstance

@singleton
class MyClass(BaseClass):
    pass
```

Pros

- Decorators are additive in a way that is often more intuitive than multiple inheritance.

Cons

- While objects created using `MyClass()` would be true singleton objects, `MyClass` itself is a function, not a class, so you cannot call class methods from it. Also for

  ```
  x = MyClass();
  y = MyClass();
  t = type(n)();
  ```

then `x == y` but `x != t && y != t`

Method 2: A base class

```
class Singleton(object):
    _instance = None
    def __new__(class_, *args, **kwargs):
        if not isinstance(class_._instance, class_):
            class_._instance = object.__new__(class_, *args, **kwargs)
        return class_._instance

class MyClass(Singleton, BaseClass):
    pass
```

Pros

- It's a true class

Cons

- Multiple inheritance - eugh! `__new__` could be overwritten during inheritance from a second base class? One has to think more than is necessary.

Method 3: A metaclass

```
class Singleton(type):
    _instances = {}
    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super(Singleton, cls).__call__(*args,
**kwargs)
        return cls._instances[cls]

#Python2
class MyClass(BaseClass):
    __metaclass__ = Singleton

#Python3
class MyClass(BaseClass, metaclass=Singleton):
    pass
```

Pros

- It's a true class
- Auto-magically covers inheritance
- Uses `__metaclass__` for its proper purpose (and made me aware of it)

Cons

- Are there any?

Method 4: decorator returning a class with the same name

```
def singleton(class_):
    class class_w(class_):
        _instance = None
        def __new__(class_, *args, **kwargs):
            if class_w._instance is None:
                class_w._instance = super(class_w,
                                    class_).__new__(class_,
                                                    *args,
                                                    **kwargs)
                class_w._instance._sealed = False
            return class_w._instance
        def __init__(self, *args, **kwargs):
            if self._sealed:
                return
            super(class_w, self).__init__(*args, **kwargs)
            self._sealed = True
    class_w.__name__ = class_.__name__
    return class_w

@singleton
```

```
class MyClass(BaseClass):
    pass
```

Pros

- It's a true class
- Auto-magically covers inheritance

Cons

- Is there not an overhead for creating each new class? Here we are creating two classes for each class we wish to make a singleton. While this is fine in my case, I worry that this might not scale. Of course there is a matter of debate as to whether it aught to be too easy to scale this pattern...
- What is the point of the `_sealed` attribute
- Can't call methods of the same name on base classes using `super()` because they will recurse. This means you can't customize `__new__` and can't subclass a class that needs you to call up to `__init__`.

Method 5: a module

a module file `singleton.py`

Pros

- Simple is better than complex

Cons

- Not lazily instantiated

python singleton decorator base-class metaclass

Share Edit Follow

edited May 13 at 15:41
user17242583

asked Jul 20, 2011 at 10:47

theheadofabroom
19.5k 5 31 64

18 Another three techniques: use a module instead (often - generally, I think - this is a more appropriate pattern for Python but it depends a bit on what you're doing with it); make a single instance and deal with it instead (`foo.x` or if you insist `Foo.x` instead of `Foo().x`); use class attributes and static/class methods (`Foo.x`). – Chris Morgan Jul 20, 2011 at 10:55

18 @ChrisMorgan: If you're going to use class/static methods only, then don't bother making a class, really. – Cat Plus Plus Jul 20, 2011 at 10:57

4 @Cat: The effect is similar, however the reasons behind creating a global variable can be just about anything, including not knowing any better. Why does one create a singleton? If you have to ask you shouldn't be here. This explicitness is not only more pythonic, but makes maintenance a lot more simple. Yes singletons are syntactic sugar for globals, but then classes are syntactic sugar for a whole bunch of unsightly stuff and I don't think anyone will tell you you're always better off without them. – theheadofabroom Jul 20, 2011 at 14:22

29 The anti-signletons sentiment is cargo cult programming at its worst. Same with people hearing (few bothered to actually read) "Goto statement considered harmful" and think gotos are a sign of bad code regardless of context. – Hejazzman Nov 30, 2015 at 13:31

5 Hi, thanks for your elaborate post. I am fairly new to pattern programming and to python actually, and I am surprised that although method 2 seems to most well known one (it s everywhere), hardly ever someone mentions that despite only one object is created, **init__() is called every time Singleton() or MyClass() are used anywhere. I didn't try, but AFAIK this is true for all other methods too. This hardly seems desirable when implementing a singleton, or am i missing something? Of course the solution consists of setting an attribute to avoid performing __init** twice. Just curious – chrisvp Sep 4, 2016 at 11:52

38 Answers

Sorted by: Highest score (default)

1 2 Next

1018

+50

Use a Metaclass

I would recommend **Method #2**, but you're better off using a **metaclass** than a base class. Here is a sample implementation:

```
class Singleton(type):
    _instances = {}
    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super(Singleton, cls).__call__(*args,
**kwargs)
        return cls._instances[cls]

class Logger(object):
    __metaclass__ = Singleton
```

Or in Python3

```
class Logger(metaclass=Singleton):
    pass
```

If you want to run `__init__` every time the class is called, add

```
        else:
            cls._instances[cls].__init__(*args, **kwargs)
```

to the `if` statement in `Singleton.__call__`.

A few words about metaclasses. A metaclass is the **class of a class**; that is, a class is an **instance of its metaclass**. You find the metaclass of an object in Python with `type(obj)`. Normal new-style classes are of type `type`. `Logger` in the code above will be of type `class 'your_module.Singleton'`, just as the (only) instance of `Logger` will be of type `class 'your_module.Logger'`. When you call logger with `Logger()`, Python first asks the metaclass of `Logger`, `Singleton`, what to do, allowing instance creation to be pre-empted. This process is the same as Python asking a class what to do by calling `__getattr__` when you reference one of it's attributes by doing `myclass.attribute`.

A metaclass essentially decides **what the definition of a class means** and how to implement that definition. See for example http://code.activestate.com/recipes/498149/, which essentially recreates C-style `struct`s in Python using metaclasses. The thread What are some (concrete) use-cases for metaclasses? also provides some examples, they generally seem to be related to declarative programming, especially as used in ORMs.

In this situation, if you use your **Method #2**, and a subclass defines a `__new__` method, it will be **executed every time** you call `SubClassOfSingleton()` -- because it is responsible for calling the method that returns the stored instance. With a metaclass, it will **only be called once**, when the only instance is created. You want to **customize what it means to call the class**, which is decided by it's type.

In general, it **makes sense** to use a metaclass to implement a singleton. A singleton is special because is **created only once**, and a metaclass is the way you customize the **creation of a class**. Using a metaclass gives you **more control** in case you need to customize the singleton class definitions in other ways.

Your singletons **won't need multiple inheritance** (because the metaclass is not a base class), but for **subclasses of the created class** that use multiple inheritance, you need to make sure the singleton class is the **first / leftmost** one with a metaclass that redefines `__call__` This is very unlikely to be an issue. The instance dict is **not in the instance's namespace** so it won't accidentally overwrite it.

You will also hear that the singleton pattern violates the "Single Responsibility Principle" -- each class should do **only one thing**. That way you don't have to worry about messing up one thing the code does if you need to change another, because they are separate and encapsulated. The metaclass implementation **passes this test**. The metaclass is responsible for **enforcing the pattern** and the created class and subclasses need not be **aware that they are singletons**. **Method #1** fails this test, as you noted with "MyClass itself is a a function, not a class, so you cannot call class methods from it."

Python 2 and 3 Compatible Version

Writing something that works in both Python2 and 3 requires using a slightly more complicated scheme. Since metaclasses are usually subclasses of type `type`, it's possible to use one to dynamically create an intermediary base class at run time with it as its metaclass and then use *that* as the baseclass of the public `Singleton` base class. It's harder to explain than to do, as illustrated next:

```
# works in Python 2 & 3
class _Singleton(type):
    """ A metaclass that creates a Singleton base class when called. """
    _instances = {}
    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super(_Singleton, cls).__call__(*args,
**kwargs)
        return cls._instances[cls]

class Singleton(_Singleton('SingletonMeta', (object,), {})): pass

class Logger(Singleton):
    pass
```

An ironic aspect of this approach is that it's using subclassing to implement a metaclass. One possible advantage is that, unlike with a pure metaclass, `isinstance(inst, Singleton)` will return `True`.

Corrections

On another topic, you've probably already noticed this, but the base class implementation in your original post is wrong. `_instances` needs to be **referenced on the class**, you need to use `super()` or you're **recursing**, and `__new__` is actually a static method that you have to **pass the class to**, not a class method, as the actual class **hasn't been created** yet when it is called. All of these things will be true for a metaclass implementation as well.

```
class Singleton(object):
  _instances = {}
  def __new__(class_, *args, **kwargs):
    if class_ not in class_._instances:
        class_._instances[class_] = super(Singleton, class_).__new__(class_,
*args, **kwargs)
    return class_._instances[class_]

class MyClass(Singleton):
  pass

c = MyClass()
```

Decorator Returning A Class

I originally was writing a comment but it was too long, so I'll add this here. **Method #4** is better than the other decorator version, but it's more code than needed for a singleton, and it's not as clear what it does.

The main problems stem from the class being it's own base class. First, isn't it weird to have a class be a subclass of a nearly identical class with the same name that exists only in its `__class__` attribute? This also means that you can't define **any methods that call the method of the same name on their base class** with `super()` because they will recurse. This means your class can't customize `__new__` , and can't derive from any classes that need `__init__` called on them.

When to use the singleton pattern

Your use case is **one of the better examples** of wanting to use a singleton. You say in one of the comments "To me logging has always seemed a natural candidate for Singletons." You're **absolutely right**.

When people say singletons are bad, the most common reason is they are **implicit shared state**. While with global variables and top-level module imports are **explicit** shared state, other objects that are passed around are generally instantiated. This is a good point, **with two exceptions**.

The first, and one that gets mentioned in various places, is when the singletons are **constant**. Use of global constants, especially enums, is widely accepted, and considered sane because no matter what, **none of the users can mess them up for any other user**. This is equally true for a constant singleton.

The second exception, which get mentioned less, is the opposite -- when the singleton is **only a data sink**, not a data source (directly or indirectly). This is why loggers feel like a "natural" use for singletons. As the various users are **not changing the loggers** in ways other users will care about, there is **not really shared state**. This negates the primary argument against the singleton pattern, and makes them a reasonable choice because of their **ease of use** for the task.

Here is a quote from http://googletesting.blogspot.com/2008/08/root-cause-of-singletons.html:

> Now, there is one kind of Singleton which is OK. That is a singleton where all of the reachable objects are immutable. If all objects are immutable than Singleton has no global state, as everything is constant. But it is so easy to turn this kind of singleton into mutable one, it is very slippery slope. Therefore, I am against these Singletons too, not because they are bad, but because it is very easy for them to go bad. (As a side note Java enumeration are just these kind of singletons. As long as you don't put state into your enumeration you are OK, so please don't.)
>
> The other kind of Singletons, which are semi-acceptable are those which don't effect the execution of your code, They have no "side effects". Logging is perfect example. It is loaded with Singletons and global state. It is acceptable (as in it will not hurt you) because your application does not behave any different whether or not a given logger is enabled. The information here flows one way: From your application into the logger. Even thought loggers are global state since no information flows from loggers into your application, loggers are acceptable. You should still inject your logger if you want your test to assert that something is getting logged, but in general Loggers are not harmful despite being full of state.

Share Edit Follow

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

answered Jul 23, 2011 at 3:28
agf
165k 41 282 234

5 No, singletons are never good. Logging might be a good candidate for being a global (as terrible as they are), but certainly not singleton. – Cat Plus Plus Jul 23, 2011 at 11:32

16 Look at googletesting.blogspot.com/2008/08/.... It is generally anti-singleton (for good reason) but it has a good explaination of why immutable singletons and singletons without side effects don't have the same problems, if you're careful. I'm going to quote it a bit at the end of my post. – agf Jul 23, 2011 at 11:36

5 My problem with singletons is the stupid premise of "only one instance". That and tonne of thread safety problems. And dependency hiding. Globals are bad, and singletons are just globals with more problems. – Cat Plus Plus Jul 23, 2011 at 11:46

8 @Cat There are very good uses for singletons. Lazy instantiation of hardware modules (especially in single threaded applications) is one of them (but thread safe singletons also exist). – Paul Manta Aug 6, 2011 at 13:47

4 @Alcott `__new__` in a metaclass is when the *class* is new -- when it's defined, not when the *instance* would be new. Calling the class (`MyClass()`) is the operation you want to override, not the definition of the class. If you really want to understand how Python works, the best thing you can do (other than keep using it) is read docs.python.org/reference/datamodel.html. A good reference on metaclasses is eli.thegreenplace.net/2011/08/14/python-metaclasses-by-example. A good article on singletons is the series from the google blog I linked in this answer. – agf Sep 20, 2011 at 6:28

137

```
class Foo(object):
    pass

some_global_variable = Foo()
```

Modules are imported only once, everything else is overthinking. Don't use singletons and try not to use globals.

Share Edit Follow

answered Jul 20, 2011 at 10:52
Cat Plus Plus
123k 26 196 222

46 why did you say "Don't use singletons"? Any reason? – Alcott Sep 20, 2011 at 1:30

4 This won't work if the singleton has to be pickled. Using the example you gave: `s = some_global_variable; str = pickle.dumps(s); s1 = pickle.loads(str); print s is s1; # False` – dividebyzero Jan 20, 2012 at 11:47

15 @dividebyzero: the `is` operator tests for pointer equality. I would be rather surprised---to the point of calling it a bug---if `pickle.loads` returned a reference to a pre-existing object rather than a reference to a newly created one. Thus, testing whether `s is s1` doesn't tell you anything about the suitability of using modules as singletons. – Jonas Kölker Feb 17, 2014 at 10:52

5 @leo-the-manic: fair point; however, that's just a side effect of Python interning the objects `True` , `False` and `None` , and has nothing to do with the code behind `pickle.loads` . Also, it's safe to do only for read-only objects. If `pickle.loads` were to return a reference to an already existing *modifiable* object—such as a module—that would be a bug. (And so I'm standing by my implication that dividebyzero's code example doesn't prove anything.) – Jonas Kölker May 3, 2014 at 0:18

4 I imported the module in several different script files (all of which would use the "singleton"), with `import ABC as X` , and it was actually imported once for every script that had the `import` statement for the "singleton" module. So the variable I defined had its value reassigned each time the module was imported. I also tried removing the alias, and even importing it in every file of the project, same results. I'm on Python 3.9.6. Singleton with metaclass (from agf's answer) did the trick fantastically for my use case (logging, with a variable that must be initialized at compile time once) – Vinícius Queiroz Oct 29, 2021 at 20:05

87

Use a module. It is imported only once. Define some global variables in it - they will be singleton's 'attributes'. Add some functions - the singleton's 'methods'.

Share Edit Follow

edited Jul 25, 2011 at 18:13

answered Jul 20, 2011 at 10:58
warvariuc
55.1k 37 168 226

16 So what you end up with is... Not a class. You can't use it as a class, you can't base other classes upon it, you use import syntax, and all of a sudden you lose all the benefits of OOP... – theheadofabroom Jul 25, 2011 at 13:04

25 if you can base other classes on it, then it might not be a singleton. you could create one of the derived class, but also one of the base class, but the derived class is also a member of the base, and you have two of the base, which one are you supposed to use? – SingleNegationElimination Jul 25, 2011 at 19:32

1 @PaulKenjora You must have an error in your code. If you define a global variable in a module, when you access it from another module it should have the value. – warvariuc Mar 30, 2017 at 19:40

1 @theheadofabroom you could `import * from base_module` ... rethink OOP my friend! hahahah – polvoazul Mar 7, 2018 at 4:46

3 How could you init a singleton object with arguments in a module? – nn0p Jun 10, 2019 at 4:51

63

You probably never need a singleton in Python. Just define all your data and functions in a module and you have a de facto singleton:

```
import datetime
file_name=None

def set_file_name(new_file_name: str):
    global file_name
    file_name=new_file_name

def write(message: str):
    global file_name
    if file_name:
        with open(file_name, 'a+') as f:
            f.write("{} {}\n".format(datetime.datetime.now(), message))
    else:
        print("LOG: {}", message)
```

To use:

```
import log
log.set_file_name("debug.log")
log.write("System starting")
...
```

If you really absolutely have to have a singleton class then I'd go with:

```
class MySingleton(object):
    def foo(self):
        pass

my_singleton = MySingleton()
```

To use:

```
from mysingleton import my_singleton
my_singleton.foo()
```

where `mysingleton.py` is your filename that `MySingleton` is defined in. This works because after the first time a file is imported, Python doesn't re-execute the code.

Share Edit Follow

edited Feb 21 at 11:08
Neuron
4,730 4 35 54

answered Dec 10, 2014 at 22:24
Alan Dyke
809 6 14

12 Mostly true, but sometimes that's not enough. E.g. I have a project with a need to log instantiations of many classes at DEBUG level. I need command line options parsed at startup in order to set the user-specified logging level *before* those classes are instantiated. Module-level instantiations make that problematic. It's possible that I could carefully structure the app so that all of those classes don't get imported until the CLI processing is done, but natural structure of my app is more important than dogmatic adherence to "singletons are bad", since they can be done quite cleanly. – mikenerone Aug 27, 2016 at 21:29

Here's a one-liner for you:

15

```
singleton = lambda c: c()
```

Here's how you use it:

```
@singleton
class wat(object):
    def __init__(self): self.x = 1
    def get_x(self): return self.x

assert wat.get_x() == 1
```

Your object gets instantiated eagerly. This may or may not be what you want.

Share Edit Follow

answered Oct 19, 2013 at 14:43
Jonas Kölker
7,610 2 43 51

2 Why do you need to know use the class of a singleton? Just use the singleton object.. – Tolli Oct 17, 2014 at 23:41

1 It's not singleton pattern, so IMO the function should be named differently. – GingerPlusPlus Dec 30, 2014 at 21:53

9 Wikipedia: "the singleton pattern is a design pattern that restricts the instantiation of a class to one object". I would say that my solution does just that. Okay, I guess one could do `wat2 = type(wat)()` , but this is python, we're all consenting adults and all that. You can't *guarantee* that there will be only one instance, but you can guarantee that if people make a second one, it will look ugly and—if they're decent, upstanding people—like a warning sign to them. What am I missing? – Jonas Kölker Jan 2, 2015 at 19:35

1 if you are really looking for a one-line solution try a python's module as a singleton, which is actually a zero-line solution. – Sławomir Lenart Apr 5, 2021 at 8:54

12

Check out Stack Overflow question *Is there a simple, elegant way to define singletons in Python?* with several solutions.

I'd strongly recommend to watch Alex Martelli's talks on design patterns in python: part 1 and part 2. In particular, in part 1 he talks about singletons/shared state objects.

Share Edit Follow

edited May 23, 2017 at 11:55
Community Bot
1 1

answered Jul 20, 2011 at 11:00
Anton
2,326 15 17

4 While this is not really an answer to my question, the resources you point to are very useful. I begrudgingly give you a +1 – theheadofabroom Jul 25, 2011 at 13:08

9

- If one wants to have multiple number of instances of the same class, but only if the args or kwargs are different, one can use the third-party python package *Handy Decorators* (package `decorators`).
- Ex.
  1. If you have a class handling `serial` communication, and to create an instance you want to send the serial port as an argument, then with traditional approach won't work
  2. Using the above mentioned decorators, one can create multiple instances of the class if the args are different.
  3. For same args, the decorator will return the same instance which is already been created.

```
>>> from decorators import singleton
>>>
>>> @singleton
```

```
... class A:
...     def __init__(self, *args, **kwargs):
...         pass
...
>>>
>>> a = A(name='Siddhesh')
>>> b = A(name='Siddhesh', lname='Sathe')
>>> c = A(name='Siddhesh', lname='Sathe')
>>> a is b  # has to be different
False
>>> b is c  # has to be same
True
>>>
```

Share Edit Follow

edited Jun 15, 2020 at 12:47
buhtz
9,428 13 66 132

answered Aug 1, 2019 at 10:37
Siddhesh Suhas Sathe
364 2 6

1 Caveat: The `previous_instances` dictionary of the implementation of `@singleton` does not look thread safe. If one thread is constructing an object while another object checks the dictionary, there is a race condition there... – Jonathan Komar Nov 14, 2021 at 9:56

8

Using a function attribute is also very simple

```
def f():
    if not hasattr(f, 'value'):
        setattr(f, 'value', singletonvalue)
    return f.value
```

Share Edit Follow

answered Jul 2, 2020 at 9:35
Ruben Decrop
1,869 1 17 9

6

Here's my own implementation of singletons. All you have to do is decorate the class; to get the singleton, you then have to use the `Instance` method. Here's an example:

```
@Singleton
class Foo:
    def __init__(self):
        print 'Foo created'

f = Foo() # Error, this isn't how you get the instance of a singleton

f = Foo.Instance() # Good. Being explicit is in line with the Python Zen
g = Foo.Instance() # Returns already created instance

print f is g # True
```

And here's the code:

```
class Singleton:
    """
    A non-thread-safe helper class to ease implementing singletons.
    This should be used as a decorator -- not a metaclass -- to the
    class that should be a singleton.

    The decorated class can define one `__init__` function that
    takes only the `self` argument. Other than that, there are
    no restrictions that apply to the decorated class.

    To get the singleton instance, use the `Instance` method. Trying
    to use `__call__` will result in a `TypeError` being raised.

    Limitations: The decorated class cannot be inherited from.

    """

    def __init__(self, decorated):
        self._decorated = decorated

    def Instance(self):
        """
        Returns the singleton instance. Upon its first call, it creates a
        new instance of the decorated class and calls its `__init__` method.
        On all subsequent calls, the already created instance is returned.

        """
        try:
            return self._instance
        except AttributeError:
            self._instance = self._decorated()
            return self._instance

    def __call__(self):
        raise TypeError('Singletons must be accessed through `Instance()`.')

    def __instancecheck__(self, inst):
        return isinstance(inst, self._decorated)
```

Share Edit Follow

edited Feb 21 at 11:09
Neuron 4,730 4 35 54

answered Sep 20, 2011 at 18:04
Paul Manta 30k 31 124 204

3 It's not true singleton. `SingletonList = Singleton(list).Instance(); print(SingletonList is type(SingletonList)())` should print `True` in true singleton; with your code prints `False` – GingerPlusPlus Dec 30, 2014 at 22:07

1 @GingerPlusPlus I was aware of a few limitations, but not of the one you pointed out. Thanks for mentioning it. Unfortunately, I don't have time at the moment to thin about a solution to this. – Paul Manta Dec 30, 2014 at 22:19

5

I will recommend an elegant solution using metaclasses

```
class Singleton(type):
    # Inherit from "type" in order to gain access to method __call__
    def __init__(self, *args, **kwargs):
        self.__instance = None # Create a variable to store the object
reference
        super().__init__(*args, **kwargs)

    def __call__(self, *args, **kwargs):
        if self.__instance is None:
            # if the object has not already been created
            self.__instance = super().__call__(*args, **kwargs) # Call the
__init__ method of the subclass (Spam) and save the reference
            return self.__instance
        else:
            # if object (Spam) reference already exists; return it
            return self.__instance

class Spam(metaclass=Singleton):
    def __init__(self, x):
        print('Creating Spam')
        self.x = x
```

```
if __name__ == '__main__':
    spam = Spam(100)
    spam2 = Spam(200)
```

Output:

```
Creating Spam
```

As you can see from the output, only one object is instantiated

Share Edit Follow

answered Aug 19, 2020 at 8:42
Dorcioman
373 3 4

5

I prefer this solution which I found very clear and straightforward. It is using double check for instance, if some other thread already created it. Additional thing to consider is to make sure that deserialization isn't creating any other instances. https://gist.github.com/werediver/4396488

```
import threading


# Based on tornado.ioloop.IOLoop.instance() approach.
# See https://github.com/facebook/tornado
class SingletonMixin(object):
    __singleton_lock = threading.Lock()
    __singleton_instance = None

    @classmethod
    def instance(cls):
        if not cls.__singleton_instance:
            with cls.__singleton_lock:
                if not cls.__singleton_instance:
                    cls.__singleton_instance = cls()
        return cls.__singleton_instance


if __name__ == '__main__':
    class A(SingletonMixin):
        pass

    class B(SingletonMixin):
        pass

    a, a2 = A.instance(), A.instance()
    b, b2 = B.instance(), B.instance()

    assert a is a2
    assert b is b2
    assert a is not b

    print('a:  %s\na2: %s' % (a, a2))
    print('b:  %s\nb2: %s' % (b, b2))
```

Share Edit Follow

answered Oct 26, 2020 at 21:53
Andrei R.
146 1 2

2 As I mentioned before, we need it to make sure that, while we are performing 'if' and using the lock, some other thread haven't created this instance already en.wikipedia.org/wiki/Double-checked_locking It's a fairly popular concept to be asked at interviews :) – Andrei R. Oct 26, 2020 at 22:29

4

```
from functools import cache

@cache
class xxx:
   ....
```

Dead easy and works!

Share Edit Follow

answered Nov 3, 2021 at 19:20
socketpair
1,889 16 15

4

Use a class variable (no decorator)

By overriding the `__new__` method to return the same instance of the class. A boolean to only initialize the class for the first time:

```
class SingletonClass:
    _instance = None

    def __new__(cls, *args, **kwargs):
        # If no instance of class already exits
        if cls._instance is None:
            cls._instance = object.__new__(cls)
            cls._instance._initialized = False
        return cls._instance

    def __init__(self, *args, **kwargs):
        if self._initialized:
            return

        self.attr1 = args[0]
        # set the attribute to `True` to not initialize again
        self._initialized = True
```

Share Edit Follow

edited May 3 at 21:43

answered Sep 20, 2021 at 19:23
DEEPAK SURANA
401 2 13

3

Most recent versions of python already have the decorator that OP wants, It just is not called `singleton`, since its usage is more generic and meant for functions too:

Python 3.2+

```
from functools import lru_cache

@lru_cache(maxsize=None)
class CustomClass(object):

    def __init__(self, arg):
        print(f"CustomClass initialised with {arg}")
        self.arg = arg
```

Usage:

```
c1 = CustomClass("foo")
c2 = CustomClass("foo")
c3 = CustomClass("bar")

print(c1 == c2)
print(c1 == c3)
```

Output (notice how `foo` got printed only once):

```
>>> CustomClass initialised with foo
>>> CustomClass initialised with bar
>>> True
>>> False
```

Python 3.9+

```
from functools.cache

@cache
class CustomClass(object):
    ...
```

Share Edit Follow

edited Oct 10 at 7:29

answered Oct 7 at 14:31

ciurlaro
493 8 18

3

Method 3 seems to be very neat, but if you want your program to run in both Python 2 and Python 3, it doesn't work. Even protecting the separate variants with tests for the Python version fails, because the Python 3 version gives a syntax error in Python 2.

Thanks to Mike Watkins: http://mikewatkins.ca/2008/11/29/python-2-and-3-metaclasses/. If you want the program to work in both Python 2 and Python 3, you need to do something like:

```
class Singleton(type):
    _instances = {}
    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super(Singleton, cls).__call__(*args,
**kwargs)
        return cls._instances[cls]

MC = Singleton('MC', (object), {})

class MyClass(MC):
    pass    # Code for the class implementation
```

I presume that 'object' in the assignment needs to be replaced with the 'BaseClass', but I haven't tried that (I have tried code as illustrated).

Share Edit Follow

edited Mar 14, 2016 at 21:54
Peter Mortensen
30.6k 21 104 125

answered Jul 24, 2013 at 17:06
Tim
31 4

I'll toss mine into the ring. It's a simple decorator.

3

```
from abc import ABC

def singleton(real_cls):

    class SingletonFactory(ABC):

        instance = None

        def __new__(cls, *args, **kwargs):
            if not cls.instance:
                cls.instance = real_cls(*args, **kwargs)
            return cls.instance

    SingletonFactory.register(real_cls)
    return SingletonFactory

# Usage
@singleton
class YourClass:
    ...  # Your normal implementation, no special requirements.
```

Benefits I think it has over some of the other solutions:

- It's clear and concise (to my eye ;D).
- Its action is completely encapsulated. You don't need to change a single thing about the implementation of `YourClass`. This includes not needing to use a metaclass for your class (note that the metaclass above is on the factory, not the "real" class).
- It doesn't rely on monkey-patching anything.
- It's transparent to callers:
  - Callers still simply import `YourClass`, it looks like a class (because it is), and they use it normally. No need to adapt callers to a factory function.
  - What `YourClass()` instantiates is still a true instance of the `YourClass` you implemented, not a proxy of any kind, so no chance of side effects resulting from that.
  - `isinstance(instance, YourClass)` and similar operations still work as expected (though this bit does require abc so precludes Python <2.6).

One downside does occur to me: classmethods and staticmethods of the real class are not transparently callable via the factory class hiding it. I've used this rarely enough that I've never happen to run into that need, but it would be easily rectified by using a custom metaclass on the factory that implements `__getattr__()` to delegate all-ish attribute access to the real class.

A related pattern I've actually found more useful (not that I'm saying these kinds of things are required very often at all) is a "Unique" pattern where instantiating the class with the *same arguments* results in getting back the same instance. I.e. a "singleton per arguments". The above adapts to this well and becomes even more concise:

```
def unique(real_cls):

    class UniqueFactory(ABC):

        @functools.lru_cache(None)  # Handy for 3.2+, but use any memoization 
decorator you like
        def __new__(cls, *args, **kwargs):
            return real_cls(*args, **kwargs)

    UniqueFactory.register(real_cls)
    return UniqueFactory
```

All that said, I do agree with the general advice that if you think you need one of these things, you really should probably stop for a moment and ask yourself if you really do. 99% of the time, YAGNI.

Share Edit Follow

edited Aug 30, 2016 at 4:58

answered Aug 27, 2016 at 22:22

mikenerone
1,892 3 15 19

3

Maybe I missunderstand the singleton pattern but my solution is this simple and pragmatic (pythonic?). This code fullfills two goals

1. Make the instance of `Foo` accessiable everywhere (global).
2. Only one instance of `Foo` can exist.

This is the code.

```
#!/usr/bin/env python3

class Foo:
    me = None

    def __init__(self):
        if Foo.me != None:
            raise Exception('Instance of Foo still exists!')

        Foo.me = self


if __name__ == '__main__':
    Foo()
    Foo()
```

Output

```
Traceback (most recent call last):
  File "./x.py", line 15, in <module>
    Foo()
  File "./x.py", line 8, in __init__
    raise Exception('Instance of Foo still exists!')
Exception: Instance of Foo still exists!
```

Share Edit Follow

answered Aug 18, 2019 at 20:44

buhtz
9,428 13 66 132

2

Well, other than agreeing with the general Pythonic suggestion on having module-level global, how about this:

```
def singleton(class_):
    class class_w(class_):
        _instance = None
        def __new__(class2, *args, **kwargs):
            if class_w._instance is None:
                class_w._instance = super(class_w, class2).__new__(class2,
*args, **kwargs)
                class_w._instance._sealed = False
            return class_w._instance
        def __init__(self, *args, **kwargs):
            if self._sealed:
                return
            super(class_w, self).__init__(*args, **kwargs)
            self._sealed = True
    class_w.__name__ = class_.__name__
    return class_w
```

```
@singleton
class MyClass(object):
    def __init__(self, text):
        print text
    @classmethod
    def name(class_):
        print class_.__name__

x = MyClass(111)
x.name()
y = MyClass(222)
print id(x) == id(y)
```

Output is:

```
111     # the __init__ is called only on the 1st time
MyClass # the __name__ is preserved
True    # this is actually the same instance
```

Share Edit Follow

edited Jul 25, 2011 at 19:27

answered Jul 25, 2011 at 0:09
Guard
6,678 3 37 57

2

How about this:

```
def singleton(cls):
    instance=cls()
    cls.__new__ = cls.__call__= lambda cls: instance
    cls.__init__ = lambda self: None
    return instance
```

Use it as a decorator on a class that should be a singleton. Like this:

```
@singleton
class MySingleton:
    #....
```

This is similar to the `singleton = lambda c: c()` decorator in another answer. Like the other solution, the only instance has name of the class (`MySingleton`). However, with this solution you can still "create" instances (actually get the only instance) from the class, by doing `MySingleton()` . It also prevents you from creating additional instances by doing `type(MySingleton)()` (that also returns the same instance).

Share Edit Follow

edited Jan 2, 2015 at 5:25

answered May 17, 2014 at 23:48
Tolli
362 5 12

1 Every time you call `type(MySingleton)()` , `MySingleton.__init__()` gets called and the object gets initialized multiple Times; you can fix it writing `cls.__init__ = lambda self: pass` in your `singleton` . Also, overriding `cls.__call__` seems pointless, and even harmful - `__call__` defined in this context is used when you call `MySingleton(any, list, of, arguments)` , not when you call `type(MySingleton)(any, list, of, arguments)` . – GingerPlusPlus Dec 31, 2014 at 13:23

1 @GingerPlusPlus, Regarding the use of `__call__` . my intention was to make both `type(MySingleton)()` and `MySingleton()` return the instance. So it is doing what I wanted. You can think of MySingleton as either the type of the singleton or the instance of the singleton (or both). – Tolli Jan 2, 2015 at 5:43

This answer is likely not what you're looking for. I wanted a singleton in the sense that only that object had its identity, for comparison to. In my case it

2

was being used as a Sentinel Value. To which the answer is very simple, make any object `mything = object()` and by python's nature, only that thing will have its identity.

```
#!python
MyNone = object()  # The singleton

for item in my_list:
    if item is MyNone:  # An Example identity comparison
        raise StopIteration
```

Share Edit Follow

answered Sep 29, 2016 at 18:40
ThorSummoner
15.4k 15 130 140

2

> Pros
>
> It's a true class Auto-magically covers inheritance Uses **metaclass** for its proper purpose (and made me aware of it) Cons
>
> Are there any?

This will be problem with serialziation. If you try to deserialize object from file (pickle) it will not use `__call__` so it will create new file, you can use base class inheritance with `__new__` to prevent that.

Share Edit Follow

answered Mar 6, 2021 at 19:21
Vintage
41 7

2

I also prefer decorator syntax to deriving from metaclass. My two cents:

```
from typing import Callable, Dict, Set


def singleton(cls_: Callable) -> type:
    """ Implements a simple singleton decorator
    """
    class Singleton(cls_):  # type: ignore
        __instances: Dict[type, object] = {}
        __initialized: Set[type] = set()

        def __new__(cls, *args, **kwargs):
            if Singleton.__instances.get(cls) is None:
                Singleton.__instances[cls] = super().__new__(cls, *args,
**kwargs)
            return Singleton.__instances[cls]

        def __init__(self, *args, **kwargs):
            if self.__class__ not in Singleton.__initialized:
                Singleton.__initialized.add(self.__class__)
                super().__init__(*args, **kwargs)

    return Singleton


@singleton
class MyClass(...):
    ...
```

This has some benefits above other decorators provided:

- `isinstance(MyClass(), MyClass)` will still work (returning a function from the clausure instead of a class will make isinstance to fail)
- `property`, `classmethod` and `staticmethod` will still work as expected
- `__init__()` constructor is executed only once
- You can inherit from your decorated class (useless?) using @singleton again

Cons:

- `print(MyClass().__class__.__name__)` will return `Singleton` instead of `MyClass`. If you still need this, I recommend using a metaclass as suggested above.

If you need a different instance based on constructor parameters this solution needs to be improved (solution provided by siddhesh-suhas-sathe provides this).

Finally, as other suggested, consider using a module in python. Modules *are* objects. You can even pass them in variables and inject them in other classes.

Share Edit Follow

edited Jul 20, 2021 at 8:38

answered Dec 7, 2020 at 14:23
Boriel
141 2 6

2

I just made a simple one by accident and thought I'd share it...

```
class MySingleton(object):
    def __init__(self, *, props={}):
        self.__dict__ = props

mything = MySingleton()
mything.test = 1
mything2 = MySingleton()
print(mything2.test)
mything2.test = 5
print(mything.test)
```

Share Edit Follow

answered Aug 5, 2021 at 21:16
byteface
152 2 7

2

It is slightly similar to the answer by fab but not exactly the same.

The singleton pattern does not require that we be able to call the constructor multiple times. As a singleton should be created once and once only, shouldn't it be seen to be created just once? "Spoofing" the constructor arguably impairs legibility.

So my suggestion is just this:

```
class Elvis():
    def __init__(self):
        if hasattr(self.__class__, 'instance'):
            raise Exception()
        self.__class__.instance = self
        # initialisation code...
    @staticmethod
    def the():
        if hasattr(Elvis, 'instance'):
            return Elvis.instance
        return Elvis()
```

This does not rule out the use of the constructor or the field `instance` by user code:

```
if Elvis() is King.instance:
```

... if you know for sure that `Elvis` has not yet been created, and that `King` has.

But it *encourages* users to use the `the` method universally:

```
Elvis.the().leave(Building.the())
```

To make this complete you could also override `__delattr__()` to raise an Exception if an attempt is made to delete `instance`, and override `__del__()` so that it raises an Exception (unless we know the program is ending...)

Further improvements

My thanks to those who have helped with comments and edits, of which more are welcome. While I use Jython, this should work more generally, and be thread-safe.

```
try:
    # This is jython-specific
    from synchronize import make_synchronized
except ImportError:
    # This should work across different python implementations
    def make_synchronized(func):
        import threading
        func.__lock__ = threading.Lock()

        def synced_func(*args, **kws):
            with func.__lock__:
                return func(*args, **kws)

        return synced_func

class Elvis(object): # NB must be subclass of object to use __new__
    instance = None

    @classmethod
    @make_synchronized
    def __new__(cls, *args, **kwargs):
        if cls.instance is not None:
            raise Exception()
        cls.instance = object.__new__(cls, *args, **kwargs)
        return cls.instance

    def __init__(self):
        pass
        # initialisation code...

    @classmethod
    @make_synchronized
    def the(cls):
        if cls.instance is not None:
            return cls.instance
        return cls()
```

Points of note:

1. If you don't subclass from object in python2.x you will get an old-style class, which does not use `__new__`
2. When decorating `__new__` you must decorate with @classmethod or `__new__` will be an unbound instance method
3. This could possibly be improved by way of use of a metaclass, as this would allow you to make `the` a class-level property, possibly renaming it to `instance`

Share Edit Follow

edited Jan 21 at 14:18

answered Feb 21, 2016 at 8:48
mike rodent
13.1k 11 93 135

2

Method: override __new__ after single use

```
class Singleton():
    def __init__(self):
        Singleton.instance = self
        Singleton.__new__ = lambda _: Singleton.instance
```

Pros

- Extremely simple and concise
- True class, no modules needed
- Proper use of lambda and pythonic monkey patching

Cons

- __new__ could be overridden again

Share Edit Follow

answered May 15 at 8:10
Markus Hirsimäki
182 7

1

I can't remember where I found this solution, but I find it to be the most 'elegant' from my non-Python-expert point of view:

```
class SomeSingleton(dict):
    __instance__ = None
    def __new__(cls, *args,**kwargs):
        if SomeSingleton.__instance__ is None:
            SomeSingleton.__instance__ = dict.__new__(cls)
        return SomeSingleton.__instance__

    def __init__(self):
        pass

    def some_func(self,arg):
        pass
```

Why do I like this? No decorators, no meta classes, no multiple inheritance...and if you decide you don't want it to be a Singleton anymore, just delete the `__new__` method. As I am new to Python (and OOP in general) I expect someone will set me straight about why this is a terrible approach?

Share Edit Follow

answered Nov 11, 2014 at 5:38

2cynykyl
933 9 12

3 *why this is a terrible approach?* when you want create another singleton class, you have to copy & paste the `__new__` . Don't repeat yourself. – GingerPlusPlus Dec 30, 2014 at 17:56

1

Code based on Tolli's answer.

```
#decorator, modyfies new_cls
def _singleton(new_cls):
    instance = new_cls()                                              #2
    def new(cls):
        if isinstance(instance, cls):                                 #4
            return instance
        else:
            raise TypeError("I can only return instance of {}, caller wanted
{}".format(new_cls, cls))
    new_cls.__new__  = new                                            #3
    new_cls.__init__ = lambda self: None                              #5
    return new_cls


#decorator, creates new class
def singleton(cls):
    new_cls = type('singleton({})'.format(cls.__name__), (cls,), {} ) #1
    return _singleton(new_cls)


#metaclass
def meta_singleton(name, bases, attrs):
    new_cls = type(name, bases, attrs)                                #1
    return _singleton(new_cls)
```

Explanation:

1. Create new class, inheriting from given `cls`
   (it doesn't modify `cls` in case someone wants for example `singleton(list)`)
2. Create instance. Before overriding `__new__` it's so easy.
3. Now, when we have easily created instance, overrides `__new__` using method defined moment ago.
4. The function returns `instance` only when it's what the caller expects, otherwise raises `TypeError` .
   The condition is not met when someone attempts to inherit from decorated class.
5. > **If `__new__()` returns an instance of `cls` , then the new instance's `__init__()` method will be invoked** like `__init__(self[, ...])` , where self is the new instance and the remaining arguments are the same as were passed to `__new__()` .

   `instance` is already initialized, so function replaces `__init__` with function doing nothing.

See it working online

Share Edit Follow

edited May 23, 2017 at 12:02

Community Bot
1 1

answered Dec 30, 2014 at 21:47

GingerPlusPlus
5,103 1 28 51

1

If you don't need lazy initialization of the instance of the Singleton, then the following should be easy and thread-safe:

```
class A:
    instance = None
    # Methods and variables of the class/object A follow
A.instance = A()
```

This way `A` is a singleton initialized at module import.

Share Edit Follow

answered Jan 1, 2019 at 19:45
Serge Rogatch
13.1k 7 76 139

1

After struggling with this for some time I eventually came up with the following, so that the config object would only be loaded once, when called up from separate modules. The metaclass allows a global class instance to be stored in the builtins dict, which at present appears to be the neatest way of storing a proper program global.

```
import builtins

# -----------------------------------------------------------------------------
# So..... you would expect that a class would be "global" in scope, however
#   when different modules use this,
#   EACH ONE effectively has its own class namespace.
#   In order to get around this, we use a metaclass to intercept
#   "new" and provide the "truly global metaclass instance" if it already
exists

class MetaConfig(type):
    def __new__(cls, name, bases, dct):
        try:
            class_inst = builtins.CONFIG_singleton

        except AttributeError:
            class_inst = super().__new__(cls, name, bases, dct)
            builtins.CONFIG_singleton = class_inst
            class_inst.do_load()

        return class_inst

# -----------------------------------------------------------------------------

class Config(metaclass=MetaConfig):

    config_attr = None

    @classmethod
    def do_load(cls):
        ...<load-cfg-from-file>...
```

Share Edit Follow

answered Oct 18, 2019 at 14:24
Den-Jason
2,064 1 18 14

1

You can use a `metaclass` if you want to use `instance` as a property. For example;

```
class SingletonMeta(type):
    def __init__(cls, *args, **kwargs):
        super().__init__(*args, **kwargs)
        cls._instance = None
        cls._locker = threading.Lock()

    @property
    def instance(self, *args, **kwargs):
        if self._instance is None:
            with self._locker:
                if self._instance is None:
                    self._instance = self(*args, **kwargs)
        return self._instance


class MyClass(metaclass=SingletonMeta):
    def __init__(self):
        # init here
        pass


# get the instance
my_class_instance = MyClass.instance
```

Share Edit Follow

answered Mar 10, 2021 at 16:18

omerfarukdogan

839 9 26

1 2 Next