Be careful with exec and eval in Python

written on Tuesday, February 1, 2011

One of the perceived features of a dynamic programming language like Python is the ability to execute code from a string. In fact many people are under the impression that this is the main difference between something like Python and C#. That might have been true when the people compared Python to things like C. It's certainly not a necessarily a feature of the language itself. For instance Mono implements the compiler as a service and you can compile C# code at runtime, just like Python compiles code at runtime.

Wait what. Python compiles? That is correct. CPython and PyPy (the implementations worth caring about currently) are in fact creating a code object from the string you pass to *exec* or *eval* before executing it. And that's just one of the things many people don't know about the exec statement. So this post aims to clean up some of the misconceptions about the *exec* keyword (or builtin function in Python 3) and why you have to be careful with using it.

This post was inspired by a discussion on reddit about the use of the *execfile* function in the web2py web framework but also applies to other projects. Some of this here might actually not affect web2py at all and is just a general suggestion of how to deal with *exec*. There are some very good reasons for using *exec* when that's the right thing to do.

Disclaimer beforehand: the numbers for this post are taken from Python 2.7 on OS X. Do not ever trust benchmarks, take them only as a reference and test it for yourself on your target environment. Also: "Yay, another post about the security implications of *eval*/*exec*." Wrong! I am assuming that everybody already knows how to *properly* use these two, so I will not talk about security here.

Behind the Scenes of Imports

Let's start with everybody's favourite topic: Performance. That's probably the most pointless argument against *exec*, but well, it's important to know though. But before that, let's see what Python roughly does if you import a module (`import foo`):

1. it locates the module (surprise). That happens by traversing the *sys.path* info in various ways. There is builtin import logic, there are import hooks and all in all there is a lot of magic involved I don't want to go into. If you are curious, check this and this.
2. Now depending on the import hook responsible it might load bytecode (*.pyc*) or sourcecode (*.py*):
    1. If bytecode is available and the magic checksum matches the current Python interpreter's version, the timestamp of the bytecode file is newer or equal to the source version (or the source does not exist) it will load that.
    2. If the bytecode is missing or outdated it will load the source file and compile that to bytecode. For that it checks magic comments in the file header for encoding settings and decodes according to those settings. It will also check if a special tab-width comment exists to treat tabs as something else than 8 characters if necessary. Some import hooks will then generate *.pyc* files or store the bytecode somewhere else (*__pycache__*) depending on Python version and implementation.
3. The Python interpreter creates a new module object (you can do that on your own by calling *imp.new_module* or creating an instance of *types.ModuleType*. Those are equivalent) with a proper name.
4. If the module was loaded from a file the *__file__* key is set. The import system will also make sure that *__package__* and *__path__* are set properly if packages are involved before the code is executed. Import hooks will furthermore set the *__loader__* variable.
5. The Python interpreter executes the bytecode in the context of the dictionary of the module. Thus the frame locals and frame globals for the executed code are the *__dict__* attribute of that module.
6. The module is inserted into *sys.modules*.

Now first of all, none of the above steps ever passed a string to the *exec* keyword or function. That's obviously true because that happens deep inside the Python interpreter unless you are using an import hook written in Python. But even if the Python interpreter was written in Python it would never pass a string to the exec function. So what would you want to do if you want to get that string into bytecode yourself? You would use the *compile* builtin:

```
>>> code = compile('a = 1 + 2', '<string>', 'exec')
>>> exec code
>>> print a
3
```

As you can see, *exec* happily executes bytecode too. Because the *code* variable is

actually an object of type *code* and not a string. The second argument to *compile* is the filename hint. If we are compiling from an actual string there we should provide a value enclosed in angular brackets because this is what Python will do. `<string>` and `<stdin>` are common values. If you do have a file to back this up, please use the actual filename there. The last parameter is can be one of `'exec'`, `'eval'` and `'single'`. The first one is what exec is using, the second is what the *eval* function uses. The difference is that the first can contain statements, the second only expressions. `'single'` is a form of hybrid mode which is useless for anything but interactive shells. It exists solely to implement things like the interactive Python shell and is very limited in use.

Here however we were already using a feature you should never, ever, ever use: executing code in the calling code's namespace. What should you do instead? Execute against a new environment:

```
>>> code = compile('a = 1 + 2', '<string>', 'exec')
>>> ns = {}
>>> exec code in ns
>>> print ns['a']
3
```

Why should you do that? Cleaner for starters, also because exec without a dictionary has to hack around some implementation details in the interpreter. We will cover that later. For the moment: if you want to use *exec* and you plan on executing that code more than once, make sure you compile it into bytecode first and then execute that bytecode only and only in a new dictionary as namespace.

In Python 3 the `exec ... in` statement disappeared and instead you can use the new *exec* function which takes the globals and locals dictionaries as parameters.

Performance Characteristics

Now how much faster is executing bytecode over creating bytecode and executing that?:

```
$ python -mtimeit -s 'code = "a = 2; b = 3; c = a * b"' 'exec code'
10000 loops, best of 3: 22.7 usec per loop

$ python -mtimeit -s 'code = compile("a = 2; b = 3; c = a * b",
  "<string>", "exec")' 'exec code'
1000000 loops, best of 3: 0.765 usec per loop
```

32 times as fast for a very short code example. It becomes a lot worse the more code you have. Why is that the case? Because parsing Python code and converting that into Bytecode is an expensive operation compared to evaluating the bytecode. That of course also affects *execfile* which totally does not use bytecode caches, how

should it. It's not gonna magically check if there is a *.pyc* file if you are passing the path to a *foo.py* file.

Alright, lesson learned. *compile* + *exec* > *exec*. What else has to be considered when using *exec*? The next thing you have to keep in mind is that there is a huge difference between the global scope and the local scope. While both the global scope and the local scope are using dictionaries as a data storage, the latter actually is not. Local variables in Python are just pulled from the frame local dictionary and put there as necessary. For all calculations that happen between that, the dictionary is never ever used. You can quickly verify this yourself.

Execute the following thing in the Python interpreter:

```
>>> a = 42
>>> locals()['a'] = 23
>>> a
23
```

Works as expected. Why? Because the interactive Python shell executes code as part of the global namespace like any code outside of functions or class declarations. The local scope *is* the global scope:

```
>>> globals() is locals()
True
```

Now what happens if we do this at function level?

```
>>> def foo():
...  a = 42
...  locals()['a'] = 23
...  return a
...
>>> foo()
42
```

How unfortunate. No magic variable changing for us. That however is only partially correct. There is a Python opcode for synchronizing the frame dictionary with the variables from the fast local slots. There are two ways this synchronization can happen: from fast local to dictionary and the other way round. The former is implicitly done for you when you call *locals()* or access the *f_locals* attribute from a frame object, the latter happens either explicitly when using some opcodes (which I don't think are used by Python as part of the regular compilation process but nice for hacks) or when the *exec* statement is used in the frame.

So what are the performance characteristics of code executed in a global scope versus code executed at a local scope? This is a lot harder to measure because the *timeit* module does not allow us to execute code at global scope by default. So we

will need to write a little helper module that emulates that:

```
code_global = compile('''
sum = 0
for x in xrange(500000):
    sum += x
''', '<string>', 'exec')
code_local = compile('''
def f():
    sum = 0
    for x in xrange(500000):
        sum += x
''', '<string>', 'exec')

def test_global():
    exec code_global in {}

def test_local():
    ns = {}
    exec code_local in ns
    ns['f']()
```

Here we compile two times the same algorithm into a string. One time directly globally, one time wrapped into a function. Then we have two functions. The first one executes that code in an empty dictionary, the second executes the code in a new dictionary and then calls the function that was declared. Let's ask *timeit* how fast we are:

```
$ python -mtimeit -s 'from execcompile import test_global as t' 't()'
10 loops, best of 3: 67.7 msec per loop

$ python -mtimeit -s 'from execcompile import test_local as t' 't()'
100 loops, best of 3: 23.3 msec per loop
```

Again, an increase in performance [3]. Why is that? That has to do with the fact that fast locals are faster than dictionaries (duh). What is a fast local? In a local scope Python keeps track of the names of variables it knows about. Each of that variable is assigned a number (index). That index is used in an array of Python objects instead of a dictionary. It will only fall back to the dictionary if this is necessary (debugging purposes, *exec* statement used at local scope). Even though *exec* still exists in Python 3 (as a function) you no longer it at a local scope to override variables. The Python compiler does not check if the *exec* builtin is used and will not unoptimize the scope because of that [1].

All of the above knowledge is good to know if you plan on utilizing the Python interpreter to interpret your own language by generating Python code and compiling it to bytecode. That's for instance how template engines like Mako, Jinja2 or Genshi work internally in one way or another.

Semantics and Unwritten Conventions

However most people are using the *exec* statement for something else: executing actual Python code from different locations. A very popular use case is executing config files as Python code. That's for example what Flask does if you tell it to. That's usually okay because you don't expect your config file to be a place where you implement actual code. However there are also people that use *exec* to load actual Python code that declares functions and classes. This is a very popular pattern in some plugin systems and the web2py framework.

Why is that not a good idea? Because it breaks some (partially unwritten) conventions about Python code:

1. Classes and functions belong into a module. That basic rule holds for all functions and classes imported from regular modules:

   ```
   >>> from xml.sax.saxutils import quoteattr
   >>> quoteattr.__module__
   'xml.sax.saxutils'
   ```

   Why is that important? Because that is how pickle works [2]:

   ```
   >>> pickle.loads(pickle.dumps(quoteattr))
   <function quoteattr at 0x1005349b0>
   >>> quoteattr.__module__ = 'fake'
   >>> pickle.loads(pickle.dumps(quoteattr))
   Traceback (most recent call last):
     ..
   pickle.PicklingError: Can't pickle quoteattr: it's not found as fake.quoteattr
   Traceback (most recent call last):
     ..
   pickle.PicklingError: Can't pickle quoteattr: it's not found as fake.quoteattr
   ```

   If you are using *exec* to execute Python code, be prepared that some modules like pickle, inspect, pkgutil, pydoc and probably some others that depend on those will not work as expected.

2. CPython has a cyclic garbage collector, classes can have destructors and interpreter shutdown breaks up cycles. What does it mean?

   - CPython uses refcounting internally. One (of many) downsides of refcounting is that it cannot detect circular dependencies between objects. Thus Python introduced a cyclic garbage collector at one point.
   - Python however also allows destructors on objects. Destructors however mean that the cyclic garbage collector will skip these objects because it does not know in what order it should delete these objects.

Now let's look at an innocent example:

```
class Foo(object):
    def __del__(self):
        print 'Deleted'
foo = Foo()
```

Let's execute that file:

```
$ python test.py
Deleted
```

Looks good. Let's try that with exec:

```
>>> execfile('test.py', {})
>>> execfile('test.py', {})
>>> execfile('test.py', {})
>>> import gc
>>> gc.collect()
27
```

It clearly collected something, but it never collected our *Foo* instances. What the hell is happening? What's happening is that there is an implicit cycle between *foo*, and the *__del__* function itself. The function knows the scope it was created in and from *__del__* -> global scope -> *foo* instance it has a nice cycle.

Now now we know the cause, why doesn't it happen if you have a module? The reason for that is that Python will do a trick when it shuts down modules. It will override all global values that do not begin with an underscore with *None*. We can easily verify that if we print the value of *foo* instead of `'Deleted'`:

```
class Foo(object):
    def __del__(self):
        print foo
foo = Foo()
```

And of course it's *None*:

```
$ python test.py
None
```

So if we want to replicate that with *exec* or friends, we have to apply the same logic, but Python will not do that for us. If we are not careful this could lead to hard to spot memory leaks. And this is something many people rely on, because it's documented behaviour.

3. Lifetime of objects. A global namespace sticks around from when it was imported to the point where the interpreter shuts down. With *exec* you as a user no longer know when this will happen. It might happen at a random

> point before. web2py is a common offender here. In web2py the magically executed namespace comes and goes each request which is very surprising behaviour for any experienced Python developer.

Python is not PHP

Don't try to circumvent Python idioms because some other language does it differently. Namespaces are in Python for a reason and just because it gives you the tool *exec* it does not mean you should use that tool. C gives you *setjmp* and *longjmp* and yet you will be very careful with using it. The combination of *exec* and *compile* are a powerful tool for anyone that wants to implement a domain specific language on top of Python or for developers interested in *extending* (not circumventing) the Python import system.

A python developer depends on imports doing what they are documented to do and that the namespace has a specific initial value (namely that it's empty with the exception of a few internal variables such as *__name__*, *__loader__* etc.). A Python developer depends on being able to import that module by a dotted name, on the fact that modules shut down in a specific way, that they are cached in *sys.modules* etc.

Jacob Kaplan-Moss wrote a comment on Reddit about the use of *exec* in web2py a while ago which I would recommend reading. He brings up some very good points why changing the semantics of a language is a bad idea.

However web2py and it's use of *execfile* are not the only offenders in the Python web community. Werkzeug has it's fair share of abusing Python conventions as well. We were shipping (and still do) an on-demand import system which caused more problems than it solved and are currently in the progress of moving away from it (despite all the pain this is for us). Django abused Python internals as well. It was generating Python code on the fly and totally changing semantics (to the point where imports vanished without warning!). They learned their lesson as well and fixed that problem in the magic removal branch. Same goes for web.py which was abusing the *print* statement to write into an internal thread-local buffer that was then sent out as response to the browser. Also something that turned out to be a bad idea and was subsequently removed.

With that I encourage the web2py developers to reconsider their decision on the use of the *exec* statement and using regular Python modules.

Because one of the things we all have to keep in mind: if a Python developer starts his journeys in the twisted world of wrongly executed Python modules they will be very confused when they continue their travels in another Python environment.

And having different semantics in different frameworks/modules/libraries is very hurtful for Python as a runtime and language.

[1] if one wants to argue that this is obvious: it should be. But Python does track another builtin function to change the behaviour of the compiler: *super*. So it would have been possible to do the same with *exec*. It's for the better however that this does not happen.

[2] if you however set *__module__* to *None* you will notice that Python is magically still able to find your function if it originated from a module registered in *sys.modules*. How does that work? It will actually walk through *all the modules* and look at *all the global variables* to find that function again.

I have no idea who came up with that idea, but it's an incredible slow operation if a lot of modules are loaded.

[3] I actually made a mistake in this benchmark. As correctly pointed out by @thp4 the benchmark was flawed because it was comparing different iterations. This has since been fixed.

This entry was tagged python