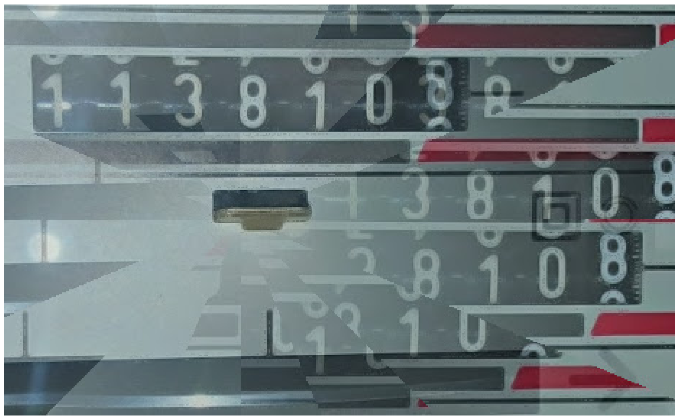
COUNT METHOD CALLS USING A METACLASS

INTRODUCTION



After you have hopefully gone through our chapter Introduction into Metaclasses you may have asked yourself about possible use cases for metaclasses. There are some interesting use cases and it's not - like some say - a solution waiting for a problem. We have mentioned already some examples.

In this chapter of our tutorial on Python, we want to elaborate an example metaclass, which will decorate the methods of the subclass. The decorated function returned by the decorator makes it possible to count the number of times each method of the subclass has been called.

This is usually one of the tasks, we expect from a profiler. So we can use this metaclass for simple profiling purposes. Of course, it will be easy to extend our metaclass for further profiling tasks.

PRELIMINARY REMARKS

Before we actually dive into the problem, we want to call to mind again how we can access the attributes of a class. We will demonstrate this with the list class. We can get the list of all the non private attributes of a class - in our example the random class - with the following construct.

```
import random
cls = "random" # name of the class as a string
all_attributes = [x for x in dir(eval(cls)) if not
x.startswith("__") ]
print(all_attributes)
```

The above Python code returned the following output:

```
['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'Random',
'SG_MAGICCONST', 'SystemRandom', 'TWOPI', '_BuiltinMethodType',
'_MethodType', '_Sequence', '_Set', '_acos', '_ceil', '_cos',
'_e', '_exp', '_inst', '_log', '_pi', '_random', '_sha512',
'_sin', '_sqrt', '_test', '_test_generator', '_urandom', '_warn',
'betavariate', 'choice', 'expovariate', 'gammavariate', 'gauss',
'getrandbits', 'getstate', 'lognormvariate', 'normalvariate',
'paretovariate', 'randint', 'random', 'randrange', 'sample',
'seed', 'setstate', 'shuffle', 'triangular', 'uniform',
'vonmisesvariate', 'weibullvariate']
```

Now, we are filtering the callable attributes, i.e. the public methods of the class.

```
methods = [x for x in dir(eval(cls)) if not x.startswith("__")
                              and callable(eval(cls + "." + x))]
print(methods)
```

This gets us the following result:

```
['Random', 'SystemRandom', '_BuiltinMethodType', '_MethodType',
'_Sequence', '_Set', '_acos', '_ceil', '_cos', '_exp', '_log',
'_sha512', '_sin', '_sqrt', '_test', '_test_generator',
'_urandom', '_warn', 'betavariate', 'choice', 'expovariate',
'gammavariate', 'gauss', 'getrandbits', 'getstate',
'lognormvariate', 'normalvariate', 'paretovariate', 'randint',
'random', 'randrange', 'sample', 'seed', 'setstate', 'shuffle',
'triangular', 'uniform', 'vonmisesvariate', 'weibullvariate']
```

Getting the non callable attributes of the class can be easily achieved by negating callable, i.e. adding "not":

```
non_callable_attributes = [x for x in dir(eval(cls)) if not
x.startswith("__")
                              and not callable(eval(cls + "." +
x))]
print(non_callable_attributes)
```

The above Python code returned the following output:

```
['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'SG_MAGICCONST',
'TWOPI', '_e', '_inst', '_pi', '_random']
```

In normal Python programming it is neither recommended nor necessary to apply methods in the following way, but it is possible:

```
lst = [3,4]
list.__dict__["append"](lst, 42)
lst
```

This gets us the following output:

```
[3, 4, 42]
```

Please note the remark from the Python documentation: "Because dir() is supplied primarily as a convenience for use at an interactive prompt, it tries to supply an interesting set of names more than it tries to supply a rigorously or consistently defined set of names, and its detailed behavior may change across releases. For example, metaclass attributes are not in the result list when the argument is a class."

A DECORATOR FOR COUNTING FUNCTION CALLS

Finally, we will begin to design the metaclass, which we have mentioned as our target in the beginning of this chapter. It will decorate all the methods of its subclass with a decorator, which counts the number of calls. We have defined such a decorator in our chapter Memoization and Decorators:

```
def call_counter(func):
    def helper(*args, **kwargs):
        helper.calls += 1
        return func(*args, **kwargs)
    helper.calls = 0
    helper.__name__= func.__name__
    return helper
```

We can use it in the usual way:

```
@call_counter
def f():
    pass
print(f.calls)
for _ in range(10):
    f()

print(f.calls)
```

The above code returned the following output:

```
0
10
```

It better if you call to mind the alternative notation for decorating function. We will need this in our final metaclass:

```
def f():
    pass
f = call_counter(f)
print(f.calls)
for _ in range(10):
    f()

print(f.calls)
```

The above code returned the following result:

```
0
10
```

THE "COUNT CALLS" METACLASS

Now we have all the necessary "ingredients" together to write our metaclass. We will include our call_counter decorator as a staticmethod:

```
class FuncCallCounter(type):
    """ A Metaclass which decorates all the methods of the
        subclass using call_counter as the decorator
    """

    @staticmethod
    def call_counter(func):
        """ Decorator for counting the number of function
            or method calls to the function or method func
        """
        def helper(*args, **kwargs):
            helper.calls += 1
            return func(*args, **kwargs)
        helper.calls = 0
        helper.__name__= func.__name__

        return helper


    def __new__(cls, clsname, superclasses, attributedict):
        """ Every method gets decorated with the decorator
call_counter,
            which will do the actual call counting
        """
        for attr in attributedict:
            if not callable(attr) and not attr.startswith("__"):
                attributedict[attr] =
cls.call_counter(attributedict[attr])

        return type.__new__(cls, clsname, superclasses,
attributedict)

class A(metaclass=FuncCallCounter):

    def foo(self):
        pass

    def bar(self):
        pass
if __name__ == "__main__":
    x = A()
    print(x.foo.calls, x.bar.calls)
    x.foo()
    print(x.foo.calls, x.bar.calls)
```

After having executed the Python code above we received the following:

```
0 0
1 0
```