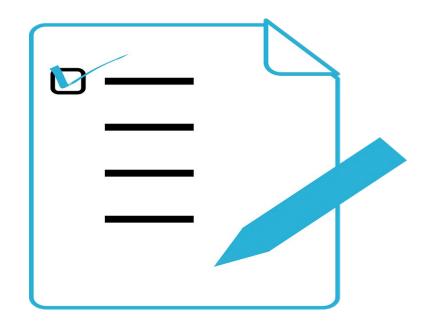
Practical Metaclasses and Decorators

Sim Zacks Principal Engineer, Red Hat 2-May-2016



Agenda

- Metaclasses
- Decorators
- Practical Usage





```
>>> class MyClass(object):
         def __init__(self):
...
              self.x = 5
...
         def add(self):
...
              self.x += 1
...
>>> mc = MyClass()
>>> type(mc)
<class '__main__.MyClass'>
>>> type(MyClass)
<type 'type'>
```

```
>>> type (5)
<type 'int'>
>>> type(type(5))
<type 'type'>
>>> type([])
<type 'list'>
>>> type(type([]))
<type 'type'>
>>> type(type)
<type 'type'>
```

```
Metaclasses
def __init__(self):
    self.x = 5
def add(self):
    self.x += 1
NewClass = type(
   "NewClass", (), {"__init__": __init__, "add": add})
>>> nc = NewClass()
>>> nc.x
>>> nc.add()
>>> nc.x
```

```
class NewType(type):
      def ___new___(cls, name, bases, attrs):
           print "%s was created" % name
           return super(NewType, cls). \
               __new__(cls, name, bases, attrs)
>>> NewClass2 = NewType("NewClass2", (),
        {"___init___": ___init___, "add": add})
NewClass2 was created
>>> nc2 = NewClass2()
>>> nc2.x
>>> nc2.add()
>>> nc2.x
```

```
class MyClass(object):
    __metaclass__ = NewType
    def __init__(self):
        self.x = 5
    def add(self):
        self.x += 1
```



```
from django.views.decorators.http \
    import require_GET
@require_GET
def my_view(request):
    # Only GET requests make it this far.
    # POSTs will fail
    pass
```

```
def my_dec(func):
    def inner(*args, **kwargs):
        return func(*args, **kwargs)
    return inner
@my_dec
def a(a, b, c):
    pass
>>> a.__name___
inner
```

```
from functools import wraps
def my_dec(func):
    @wraps(func)
    def inner(*args, **kwargs):
        print "func %s" % func.__name___
        res = func(*args, **kwargs)
    return inner
```

>>> a.__name___ a

```
>>> def f(a,b,c):
         e = 2
...
>>> f.func code.co_varnames
('a', 'b', 'c', 'd', 'e')
>>> f.func code.co varnames \
        [:f.func_code.co_argcount]
('a', 'b', 'c')
```

```
@my_dec
def a(a, b, c):
       pass
>>> "b" in a.func code.co varnames \
       [:a.func_code.co_argcount]
False
>>> a.func_code.co_varnames
('args', 'kwargs')
```

```
try:
    from decorator import decorate
except ImportError:
    raise ImportError("decorator>=4.0.9 is
        required")
def inner(func, *args, **kwargs):
def my_dec(func):
    decorate(func, inner)
```

```
>>> "b" in a.func_code.co_varnames \
... [:a.func_code.co_argcount]
True
>>> a.__name__
a
```



Requirement

Send all usage/exception data to logstash server

Environment

- Library with > 50 classes and > 10,000 lines of code
- All classes are descendants of a single base class

```
import logstash
_logger = logging.getLogger('python-logstash-logger')
_logger.setLevel(logging.INFO)
_logger.addHandler(
    logstash.LogstashHandler(
        logstash_url,
        logstash_port,
        version=1))
data = \{\}
data["XYZ"] = class_name
_logger.info(SYSTEM_NAME, extra=data)
```

```
class BaseClass()
    def send_usage_data(self, *args, **kwargs):
class A(BaseClass)
    def yadayada():
        self.send_usage_data(
            user=self.username, ...)
        ...
```

```
def send_usage_data(func):
    def inner(*arg, **kwargs)
        send_data(...)
        try:
              return func(...)
        except:
              send_error_data
              raise
```

```
class A(BaseClass)
  @send_usage_data
  def yadayadayada():
    ...
```

```
class LoggingMeta(type):
    def __new__(cls, name, bases, attrs):
        logstash_parms = {}
        for item, attr_val in attrs.items():
```

```
if isinstance(attr_val, types.FunctionType):
    attrs[item] = log_wrapper(attr_val)
elif isinstance(attr_val, classmethod):
    attrs[item] = classmethod(
         log_wrapper(attr_val.
               _get___(object).___func___))
elif item.startswith("_logstash"):
            logstash_parms[item] = attr_val
```

```
if logstash_parms:
    init_logger(**logstash_parms)
return super(LoggingMeta, cls). \
    __new__(cls, name, bases, attrs)
```

```
Practical Usage
def my_logger(func, *args, **kwargs):
    _logger.info("SYSTEM", extra=data)
    try:
        return func(*args, **kwargs)
    except Exception, e:
        data["v_errormsg"] = e
        _logger.error("SYSTEM", extra=data)
        raise
def log_wrapper(func):
    return decorate(func, my_logger)
```

```
Class BaseClass(object):
    __metaclass__ = LoggingMeta
    _logstash_url = ...
...
```

Metaclass Example Code

https://github.com/simzacks/random
_code/blob/master/logstasher.py





THANK YOU

 $g_{\text{+}}$ plus.google.com/+RedHat

Кеанат

facebook.com/redhatinc

in linkedin.com/company/red-hat

7

twitter.com/RedHatNews



youtube.com/user/RedHatVideos

Q&A



We're Looking for you! https://www.redhat.com/en/jobs

Contact me: sim@redhat.com