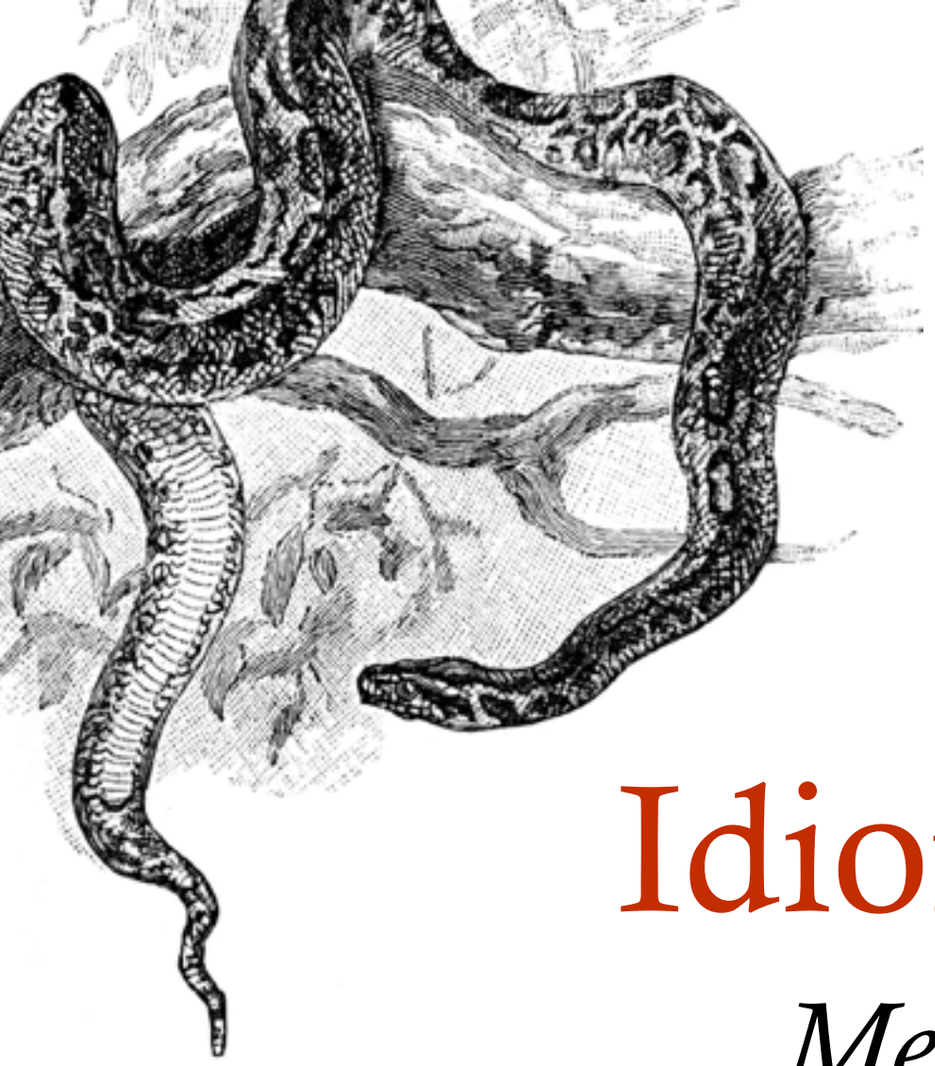


Idiomatic Python

Metaclasses and Types

Richard E Sarkis
Rochester's Python User Group
January 21st, 2014 Meeting

Expectations

- Related keywords:
  - `type`
  - `__metaclass__`
  - `__new__`
  - `__init__`
  - `__call__`

Typifying types

- `type()`
  - Determines type of an object
    - Everything is an object, including classes (so, they have a type as well)
- Examples…

Typed `type`

- The interesting case is **`type(float)`**
  - What is **`<class 'type'>`**?
  - What happens with: **`type(type)`**
    - A metaclass, as it produce a class
    - It is the default
- Printing the class of a class produces its metaclass

`type`'s second form

- **`type()`** has a second form, taking three arguments: **`type(name, bases, dict)`**
  - Creates a **new type**, programmatically

Metaprogramming

- Our defined class Foo can be made equivalently with **`type`**

```
class Foo(object):
    pass
```

- The **`type`** variant automatically subclasses from **`object`**

```
Foo = type('Foo', (), {})
```

Metaprogramming: Adding attributes

- What about member functions to `Foo`?

```
def always_false(self):
    return False

Foo.always_false = always_false
```

Metaprogramming: Adding attributes

- Or, do it in one step:

```
Foo = type('Foo', (), {'always_false': always_false})
```

So, where are metaclasses?

Metaclasses

- Metaclasses can be user defined classes
  - A class-level attribute **`__metaclass__`** is set to a callable, i.e. a function
    - This callable needs to return a type (a class), not instances
- You can provide a replacement for **`type()`** that adds extra logic in class creation.

Metaclasses: `__init__()`

- **`__init__()`** is called after the class, before it is returned to the caller
  - **`cls`** is used rather then **`self`** as the first argument
  - Practice of calling base-class constructor first, via **`super()`** should be followed:

```
super(SimpleMeta1, cls).__init__(name, bases, nmspc)
```

```
class SimpleMeta1(type):
    def __init__(cls, name, bases, nmspc):
        super(SimpleMeta1, cls).__init__(name, bases, nmspc)
        cls.uses_metaclass = lambda self : "Yes!"
```

```
class Simple1(object):
    __metaclass__ = SimpleMeta1
def foo(self): pass
@staticmethod
def bar(): pass
```

```
class SimpleMeta1(type):
    def __init__(cls, name, bases, nmspc):
        super(SimpleMeta1, cls).__init__(name, bases, nmspc)
        cls.uses_metaclass = lambda self : "Yes!"

class Simple1(object):
    __metaclass__ = SimpleMeta1
    def foo(self): pass
    @staticmethod
    def bar(): pass

simple = Simple1()
print([m for m in dir(simple) if not m.startswith('__')])
# A new method has been injected by the metaclass:
print simple.uses_metaclass()

""" Output:
['bar', 'foo', 'uses_metaclass']
Yes!
"""
```

Metaclasses: `__new__()`

- **`__new__()`** is called for the creation of the new class
  - **`cls`** is used rather then **`self`** as the first argument
  - Practice of calling base-class constructor first, via **`super()`** should be followed:

```
super(MetaBase, mcl).__new__(mcl, name, bases, nmspc)
```

```
class MyMeta(type):
    def __new__(meta, name, bases, dct):
        print '-----------------------------------'
        print "Allocating memory for class", name
        print meta
        print bases
        print dct
        return super(MyMeta, meta).__new__(meta, name, bases, dct)
```

```
class MyKlass(object):
    __metaclass__ = MyMeta

    def foo(self, param):
        pass

barattr = 2
```

```python
class MyMeta(type):
    def __new__(meta, name, bases, dct):
        print '-----------------------------------'
        print "Allocating memory for class", name
        print meta
        print bases
        print dct
        return super(MyMeta, meta).__new__(meta, name, bases, dct)

class MyKlass(object):
    __metaclass__ = MyMeta

    def foo(self, param):
        pass

    barattr = 2

""" Output:
-----------------------------------
Allocating memory for class MyKlass
<class '__main__.MyMeta'>
(<type 'object'>,)
{'barattr': 2, '__module__': '__main__',
 'foo': <function foo at 0x00B502F0>,
 '__metaclass__': <class '__main__.MyMeta'>}
"""
```

Is there a difference?

A matter of style

- **`__init__()`** has some limitations over using **`__new__()`**
- You can always use **`__new__()`** effectively
  - Can modify *name*, *bases* and *namespace* arguments before calling the **`super()`** constructor.
  - **`__init__()`** can't do this, no results from constructor call

Metaclasses: `__call__()`

- We can override object creation with __new__()
- We can override object initialization with __init__()
- What does __call__() do? It overrides the calling of the class to create an instance

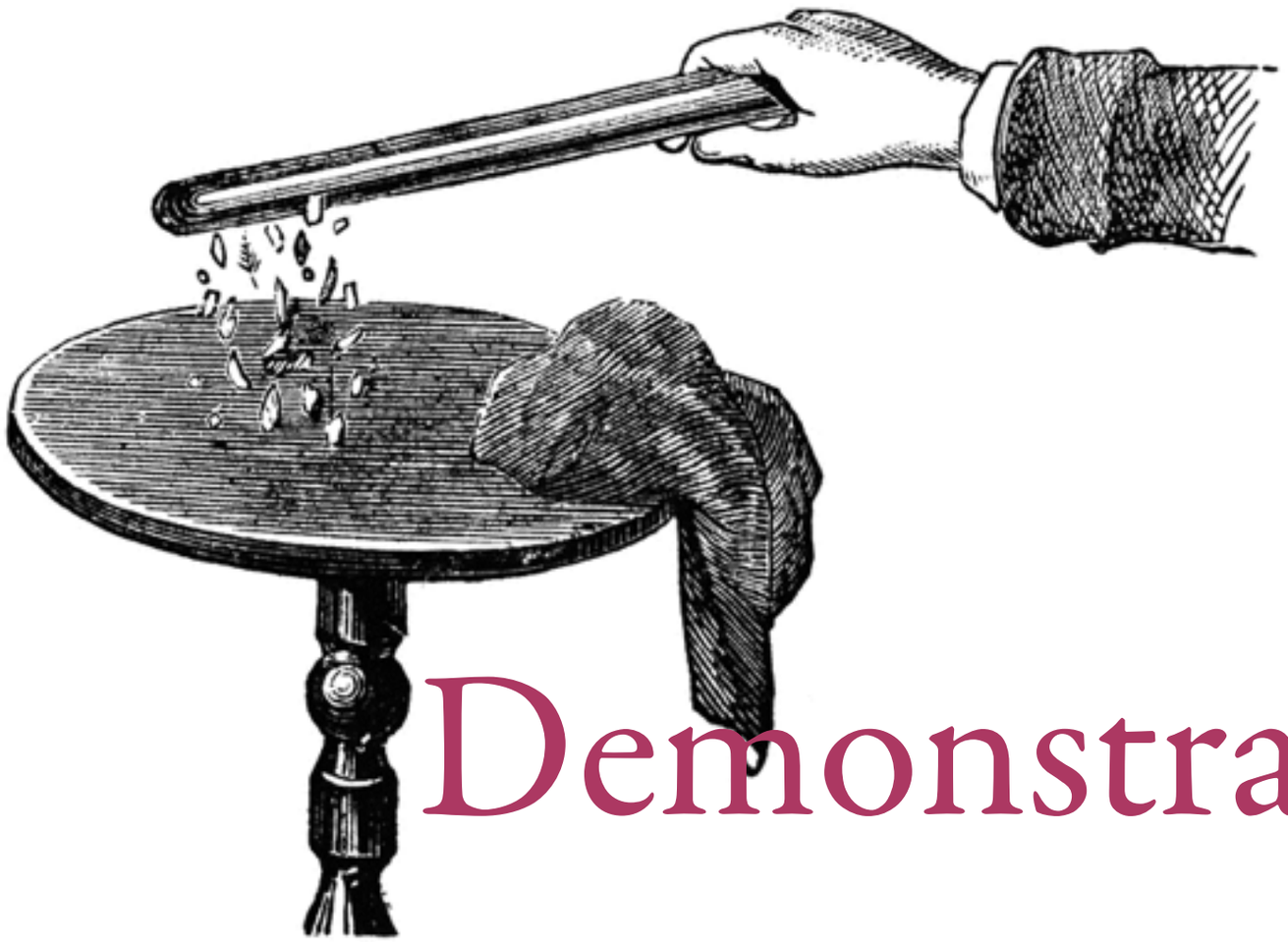
```
class MyMeta(type):
    def __call__(cls, *args, **kwds):
        print '__call__ of ', str(cls)
        print '__call__ *args=', str(args)
        return type.__call__(cls, *args, **kwds)
```

```
class MyKlass(object):
    __metaclass__ = MyMeta

    def __init__(self, a, b):
        print 'MyKlass object with a=%s, b=%s' % (a, b)
```

```python
class MyMeta(type):
    def __call__(cls, *args, **kwds):
        print '__call__ of ', str(cls)
        print '__call__ *args=', str(args)
        return type.__call__(cls, *args, **kwds)

class MyKlass(object):
    __metaclass__ = MyMeta

    def __init__(self, a, b):
        print 'MyKlass object with a=%s, b=%s' % (a, b)

print 'gonna create foo now...'
foo = MyKlass(1, 2)

""" Output:
gonna create foo now...
__call__ of  <class '__main__.MyKlass'>
__call__ *args= (1, 2)
MyKlass object with a=1, b=2
"""
```

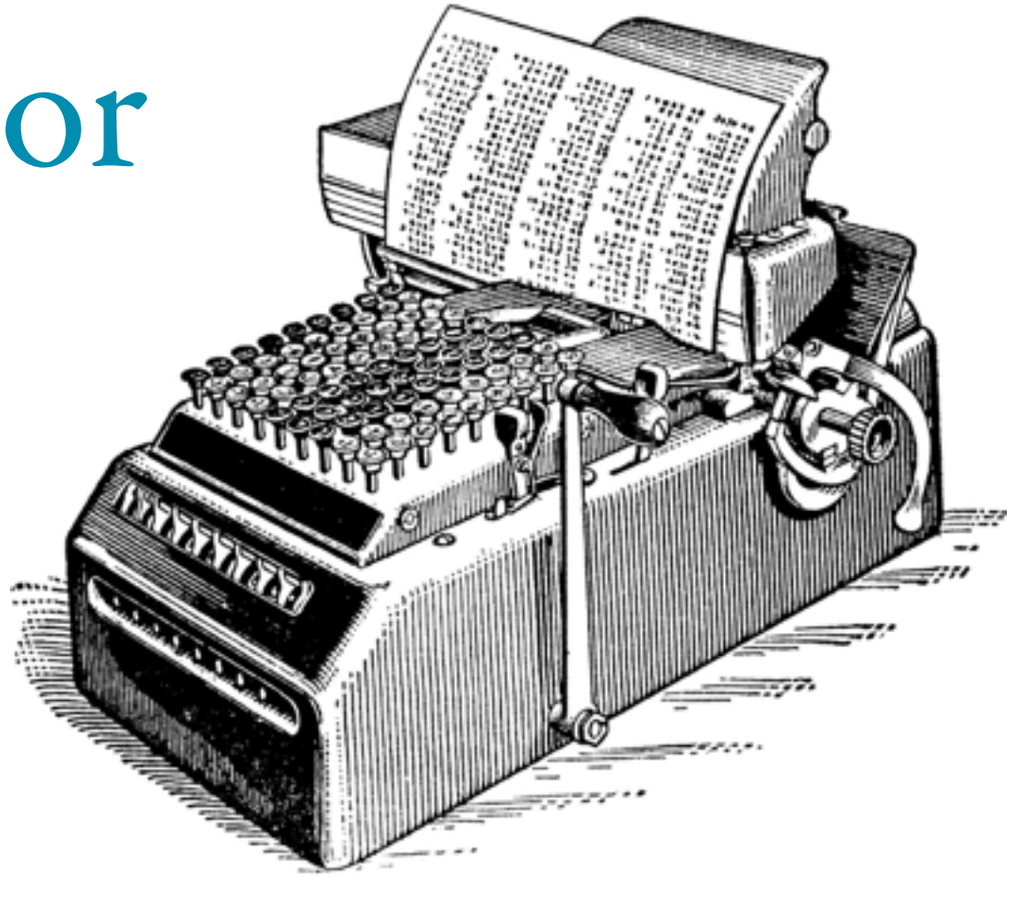
Usefulness

- Not used very often, but good to understand
  - StackOverflow answers unequivocally state there are no concrete use cases
- It can be a fun feature to play with, but it can be overwhelming to overuse it



Demonstration

Vernacular App: Calculator



Conclusion

Q & A