Meta Programming - Some Use Cases for Everyday Programming

EuroPython 2013

July 4, 2013 - Florence, Italy

Author: Dr.-Ing. Mike Müller
Email: mmueller@python-academy.de

Motivation

- Python offers powerful meta programming techniques
- Metaclasses
- Decorators for functions and classes
- Descriptors to some degree
- Dynamic code generation
- AST

Metaclasses

> ... are deeper magic than 99% of users should ever worry about. If you wonder whether you need them, you don't (the people who actually need them know with certainty that they need them, and don't need an explanation about why).
>
> Tim Peters (c.l.p post 2002-12-22)

Anatomy of Metaclass

- Metaclasses are to classes what classes are to instances
- That's it
- OK, some more details

Create a New Class

```
>>> class C(object):
        x = 1
```

- The same result but you can do at run time:

```
>>> C = type('C', (object,), dict(x=1))
```

Add Methods Too

```
>>> def __init__(self, a):
...     self.a = a
...

>>> def add(self, a, b):
...     return a + b
...

>>> C = type('C', (object,), dict(__init__=__init__, add=add))
>>> c = C(100)
>>> c.a
100
>>> c.add(30, 40)
70
```

Defining Our Own Metaclass

```
>>> class NewMeta(type):
...     def __str__(cls):
...         return 'This class is of type %s and has name %s.' \
                   %(type(cls), cls.__name__)
...
```

Making a Class From It

```
>>> # Python 3
>>> class C(metaclass=NewMeta):
...     pass


>>> # Python 2
>>> class C:
...     __metaclass__ = NewMeta
```

Having a New Class

If we print it we get the message we defined with `__str__`:

```
>>> print(C)
This class is of type <class '__main__.NewMeta'> and has name C.
```

The type of an instance of this class tells the same:

```
>>> c = C()
>>> print(type(c))
This class is of type <class '__main__.NewMeta'> and has name C.
```

Overriding __new__

- Before the class exists
- Takes a metaclass as first argument
- Returns a new class

```
>>> class MyMeta(type):
...     def __new__(mcl, name, bases, cdict):
...         print 'mcl', mcl
...         print 'name', name
...         print 'bases', bases
...         print 'class dict', cdict
...         return super(MyMeta, mcl).__new__(mcl, name, bases, cdict)
...
```

Use New Class

```
>>> class MyClass: # Python 3 MyClass(metclass=Meta)
...     __metaclass__ = MyMeta
...
mcl <class '__main__.MyMeta'>
name MyClass
bases ()
dict {'__module__': '__main__',
      '__metaclass__': <class '__main__.MyMeta'>}
```

Overriding `__init__`

- Class already exits
- Takes a class as first argument
- No return (`None`)

```
>>> class MyMeta(type):
...     def __init__(cls, name, bases, cdict):
...         print 'cls', cls
...         print 'name', name
...         print 'bases', bases
...         print 'class dict', cdict
...         super(MyMeta, cls).__init__(name, bases, cdict)
```

Use New Class

```
>>> class MyClass:
...     __metaclass__ = MyMeta
...
self <class '__main__.MyClass'>
name MyClass
bases ()
dict {'__module__': '__main__',
      '__metaclass__': <class '__main__.MyMeta'>}
```

What Can You With This?

- Possible to change the behavior of a class
- Adapt Python totally to your needs
- Danger of creating something nobody understands 6 months later (including the author ;-))

To Use Or Not To Use?

- Powerful tool
- With great power comes great responsibility
- Use with care
- Use only when it is really the best solution

Use Cases

- Production use cases
- User-facing API
- Use as a development tool

Use Case: Framework

- SQLAlchemy - ORM
- Web frameworks

Use Case: DSL

- Change the language
- Limit attributes
- Only certain methods names allowed
- Automatic operator overloading

Use Case: Development Tool

- Investigate legacy code
- Debug large code basis with very little extra code
- Monitor programs in a production environment

Use Case Monitoring / Debugging - I

- Count how many classes are defined and store their names

```
# file: autometa_python2.py

"""Example usage of a metaclass.

We change the metaclass of classes that inherit form `object`.
"""

import __builtin__

# Set this to False to deactivate DebugMeta.
DEBUG = True
```

Use Case Monitoring / Debugging - II

```
if DEBUG:
    class DebugMeta(type):
        """Metaclass to be used for debugging.

        """
        names = []
        counter = 0

        def __init__(cls, name, bases, cdict):
            """Store all class names and count how many classes are defined.
            """
            if DebugMeta.counter:
                DebugMeta.names.append('%s.%s' % (cls.__module__, name))
                print 'Debug metaclass in action. %d' % DebugMeta.counter
                print DebugMeta.names
                super(DebugMeta, cls).__init__(name, bases, cdict)
            DebugMeta.counter += 1
```

Use Case Monitoring / Debugging - III

- Replace the built-in `object` with it:

```
class new_object(object):
    """Replacement for the builtint `object`.
    """
    __metaclass__ = DebugMeta

# We actually change a builtin.
# This is a very strong measure.
__builtin__.object = new_object
```

- This is **only** for experimenting
- **Think twice** before using in production code

Use Case Monitoring / Debugging - IV

- Define some classes

```
class SomeClass1(object):
    """Test class.
    """
    pass


class SomeClass2(object):
    """Test class.
    """
    pass


class SomeClass3():
    """Test class. Does NOT inherit from object.
    """
    pass
```

Use Case Monitoring / Debugging - V

Output:

```
Debug metaclass in action. 1
['__main__.SomeClass1']
Debug metaclass in action. 2
['__main__.SomeClass1', '__main__.SomeClass2']
```

Use Case Monitoring / Debugging - VI

- Setting `DEBUG` to `False` will turn off all printing to the screen.
- Importing changes the global `object`

```
import sys


if sys.version_info[0] == 2:
    import autometa_python2 as autometa
else:
    import autometa_python3 as autometa


class MaybeDebug(object):
    """Class that may have a different metaclass.
    """
    pass
```

Use Case Monitoring / Debugging - VII

- There are a few changes when using Python 3:

```
import builtins
```

```
    class new_object(metaclass=DebugMeta):
        """Python 3 class with metaclass keyword argument.
        """
        pass
```

Uses

- Count how many times methods are called
- Measure run time of methods that match patterns in their name
- Extend logging
- Triggers for sending emails to admins
- Add your own

Danger

- Works only for new-style classes
- Make sure not to change state anywhere
- Class decorators can replace metaclasses for many tasks

Recommendations

- Use meta-programming only if no other, simple solution exists
- Try to avoid it for code you release
- Try to make your code behave as pythonic as possible
- No surprises
- **Use meta-programming for meta tasks**
- Use the right tool for the right purpose

Contact

email: mmueller@python-academy.de

twitter: @pyacademy

web: www.python-academy.com