COLUMNS

In Praise of Metaclasses!

DAVID BEAZLEY

David Beazley is an open source developer and author of the *Python Essential Reference* (4th Edition, Addison-Wesley, 2009). He is also known as the creator of Swig (http://www.swig.org) and Python Lex-Yacc (http://www.dabeaz.com/ply.html). Beazley is based in Chicago, where he also teaches a variety of Python courses. dave@dabeaz.com

Much maligned and misunderstood, metaclasses might be one of Python's most useful features. On the surface, it might not be clear why this would be the case. Just the name "metaclass" alone is enough to conjure up an image of swooping manta rays and stinging bats attacking your coworkers in a code review. I'm sure that there are also some downsides, but metaclasses really are a pretty useful thing to know about for all sorts of problems of practical interest to systems programmers. This includes simplifying the specification of network protocols, parsers, and more. In this installment, we'll explore the practical side of metaclasses and making Python do some things you never thought possible. Note: This article assumes the use of Python 3.

When I was first learning Python 20 years ago, I remember taking a trip to attend the Python conference. At that time, it was a small affair with just 50 or 60 enthusiastic programmers. I also remember one presentation in particular—the one that proposed the so-called "metaclass hack" for Python. There were a lot of frightened stares during that presentation and to be honest, it didn't make a whole lot of sense to me at the time. Some short time later, metaclasses became known as Python's "killer joke" in reference to a particular Monty Python sketch. Nobody was able to understand them without dying apparently.

Flash forward to the present and I find myself at home writing some Python code to interact with the game Minecraft. I'm buried in a sea of annoying low-level network protocol details. The solution? Metaclasses. In an unrelated project, I find myself modernizing some parsing tools I wrote about 15 years ago. Once again, I'm faced with a problem of managing lots of fiddly details. The solution? Metaclasses again. Needless to say, I'm thinking that metaclasses are actually kind of cool—maybe even awesome.

That said, metaclasses have never really been able to shake their "killer joke" quality in the Python community. They involve defining objects with the "class" statement, and inheritance is involved. Combine that with the word "meta" and surely it's just going to be some kind of icky object-oriented monstrosity birthed from the bowels of a Java framework or something. This is really too bad and misses the point.

In this article, I'm going to take a stab at rectifying that situation. We'll take a brief look at what happens when you define a class in Python, show what a metaclass is, and describe how you can use this newfound knowledge to practical advantage with an example.

Defining Classes

Most Python programmers are familiar with the idea of defining and using a class. One use of classes is to help you organize code by bundling data and functions together. For example, instead of having separate data structures and functions like this:

```
p = { 'x': 2, 'y': 3 }
def move(p, dx, dy):
    p['x'] += dx
    p['y'] += dy
```

a class lets you glue them together in a more coherent way:

```
class Point(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def move(self, dx, dy):
        self.x += dx
        self.y += dy
```

Another use of classes is as a code-reuse tool. This is common in libraries and frameworks. For example, a library will provide a base set of code for you to use and then you extend it with your own functionality via inheritance. For example, here is some code using the `socketserver` module in the standard library:

```
from socketserver import TCPServer, BaseRequestHandler

class EchoHandler(BaseRequestHandler):
    def handle(self):
        while True:
            data = self.request.recv(1024)
            if not data:
                break
            self.request.sendall(data)

serv = TCPServer(('', 25000), EchoHandler)
serv.serve_forever()
```

There is a third use of classes, however, that is a bit more interesting. Step back for a moment and think about what's happening when you define a class. Essentially, a class serves as an enclosing environment for the statements that appear inside. Within this environment, you actually have a lot of control over how Python behaves—you can bend the rules and make Python do things that are not normally possible. For example, altering definitions, validating code, or building little domain-specific languages. A good example can be found in defining an enum in the standard library [1]. Here is an example:

```
from enum import Enum

class State(Enum):
    OPEN = 1
    CLOSED = 2
```

If you start using this class and start thinking about it, you'll find that it has some unusual behavior. For example, the class variables `OPEN` and `CLOSED` that were defined as integers no longer possess those types:

```
>>> State.OPEN
<State.OPEN: 1>
>>> type(State.OPEN)
<enum 'State'>
>>> isinstance(State.OPEN, int)
False
>>>
```

Something has implicitly altered the class body in some way. You'll also find that `Enum` classes don't allow duplicate definitions. For example, this produces an error:

```
class State(Enum):
    OPEN = 1
    CLOSED = 2
    OPEN = 3

Traceback (most recent call last):
...
TypeError: Attempted to reuse key: 'OPEN'
```

If you give different names to the same value, you get an alias.

```
class State(Enum):
    OPEN = 1
    CLOSED = 2
    SHUTDOWN = 2

>>> State.CLOSED
<State.CLOSED: 2>
>>> State.SHUTDOWN
<State.CLOSED: 2>
>>> State.CLOSED is State.SHUTDOWN
True
>>>
```

If you try to inherit from an enumeration, you'll find that it's not allowed:

```
class NewState(State):
    PENDING = 3
Traceback (most recent call last):
...
TypeError: Cannot extend enumerations
```

Finally, attempting to create instances of an Enum results in a kind of type-cast rather than the creation of a new object. For example:

```
>>> s = State(2)
>>> s
<State.CLOSED>
>>> s is State.CLOSED
True
>>>
```

So *something* is not only changing the body of the class, it's monitoring the definition process itself. It's bending the normal rules of assignment. It's looking for errors and enforcing rules. Even the rules of instance creation and memory allocation have apparently changed.

These unusual features of `Enum` are an example of a metaclass in action—metaclasses are about changing the very meaning of a class definition itself. A metaclass can make a class do interesting things all while hiding in the background.

Metaclasses

Now that we've seen an example of a metaclass in action, how do you plug into this machinery yourself? The key insight is that a class definition is itself an instance of an object called `type`. For example:

```
class Spam(object):
    def yow(self):
        print('Yow!')

>>> type(Spam)
<class 'type'>
>>>
```

The *type* of a class is its metaclass. So `type` is the metaclass of `Spam`. This means that `type` is responsible for everything associated with the definition of the `Spam` class.

Now suppose you wanted to alter what happens in class creation? Here are the neat, head-exploding tricks that you can use to hook into it. This is going to look rather frightening at first, but it will make much more sense once you try it afterwards. Official documentation on the process can be found at [2].

```
class mytype(type):

    @classmethod
    def __prepare__(meta, clsname, bases):
        print('Preparing class dictionary:', clsname, bases)
        return super().__prepare__(clsname, bases)

    @staticmethod
    def __new__(meta, clsname, bases, attrs):
        print('Creating class:', clsname)
        print('Bases:', bases)
        print('Attributes:', list(attrs))
        return super().__new__(meta, clsname, bases, attrs)

    def __init__(cls, clsname, bases, attrs):
        print('Initializing class:', cls)
        super().__init__(clsname, bases, attrs)

    def __call__(cls, *args, **kwargs):
        print('Creating an instance of', cls)
       return super().__call__(*args, **kwargs)
```

In this code, we've subclassed `type` and installed hooks onto a few important methods that will be described shortly. To use this new type as a metaclass, you need to define a new top-level object like this:

```
# Top-level class
class myobject(metaclass=mytype):
    pass
```

After you've done that, using this new metaclass requires you to inherit from `myobject` like this:

```
class Spam(myobject):
    print('—Starting:', locals())
    def yow(self):
        print('Yow!')
    print('—Ending:', locals())
```

When you do this, you're going to see output from the various methods:

```
Preparing class dictionary: Spam (<class '__main__.myobject'>,)
—Starting: {'__qualname__': 'Spam', '__module__': '__main__'}
—Ending: {'__qualname__': 'Spam', '__module__': '__main__',
'yow': <function Spam.yow at 0x10e6cc9d8>}
Creating class: Spam
Bases: (<class '__main__.myobject'>,)
Attributes: ['__qualname__', '__module__', 'yow']
Initializing class: <class '__main__.Spam'>
```

Keep in mind, you have not created an instance of `Spam`. All of this is triggered automatically merely by the *definition* of the `Spam` class. An end user will see that the class `Spam` is using inheritance, but the use of a metaclass is not apparent in the specification. Let's talk about the specifics.

Before anything happens at all, you will see the `__prepare__()` method fire. The purpose of this method is to create and prepare the dictionary that's going to hold class members. This is the same dictionary that `locals()` returns in the class body. But how does Python know to use the `__prepare__()` method of our custom type? This is determined by looking at the type of the parent of `Spam`. In this case `myobject` is the parent, so this is what happens:

```
>>> ty = type(myobject)
>>> ty
<class 'meta.mytype'>
>>> d = ty.__prepare__('Spam', (myobject,))
Preparing class dictionary: Spam (<class '__main__.myobject'>,)
>>> d
{}
>>>
```

Once the class dictionary has been created, it's populated with a few bits of name information, including the class name and enclosing module.

```
>>> d['__qualname__'] = 'Spam'
>>> d['__module__'] == __name__
>>>
```

Afterwards, the body of the `Spam` class executes in this dictionary. You will see new definitions being added. Upon conclusion, the dictionary is fully populated with definitions. The print statements in the top and bottom of the class are meant to show the state of the dictionary and how it changes.

After the class body has executed, the `__new__()` method of the metaclass is triggered. This method receives information about the class, including the name, bases, and populated class dictionary. If you wanted to write code that did anything with this data prior to creating the class, this is the place to do it.

After `__new__()` is complete, the `__init__()` method fires. This method is given the newly created class as an argument. Again, this is an opportunity to change parts of the class. The main difference between `__new__()` and `__init__()` is that `__new__()` executes prior to class creation, `__init__()` executes after class creation.

The `__call__()` method of a metaclass concerns instance creation. For example:

```
>>> s = Spam()
Creating an instance of <class '__main__.Spam'>
>>> s.yow()
Yow!
>>>
```

"Yow" is right! You have just entered a whole new realm of magic. The key idea is that you can put your fingers on the knobs of class definition and instance creation—and you can start turning the knobs. Let's do it.

Example: Building a Text Tokenizer

Let's say you were building a text parser or compiler. One of the first steps of parsing is to tokenize input. For example, suppose you had an input string like this:

```
text = 'a = 3 + 4 * 5'
```

And you wanted to tokenize it in a sequence of tuples like this:

```
[ ('NAME', 'a'), ('ASSIGN', '='), ('NUM', 3),
  ('PLUS', '+'), ('NUM', 4), ('TIMES', '*'), ('NUM', 5) ]
```

One way to do this is write low-level code using regular expressions and the `re` module. For example:

```
# tok.py

import re

# Patterns for the different tokens
NAME = r'(?P<NAME>[a-zA-Z_][a-zA-Z0-9_]*)'
NUM = r'(?P<NUM>\d+)'
ASSIGN = r'(?P<ASSIGN>=)'
PLUS = r'(?P<PLUS>\+)'
TIMES = r'(?P<TIMES>\*)'
ignore = r'(?P<ignore>\s+)'

# Master re pattern
pat = re.compile('|'.join([NAME, NUM, ASSIGN, PLUS, TIMES,
ignore]))

# Tokenization function
def tokenize(text):
    index = 0
    while index < len(text):
        m = pat.match(text, index)
        if m:
            tokname = m.lastgroup
            toktext = m.group()
            if tokname != 'ignore':
                yield (tokname, toktext)
            index = m.end()
        else:
            raise SyntaxError('Bad character %r' % text[index])

if __name__ == '__main__':
    text = 'a = 3 + 4 * 5'
    for tok in tokenize(text):
        print(tok)
```

Although there's not a lot of code, it's kind of low-level and nasty looking. For example, having to use named regex groups, forming the master pattern, and so forth. Let's look at a completely different formulation using metaclasses. Define the following metaclass:

```
from collections import OrderedDict
import re

class tokenizemeta(type):
    @classmethod
    def __prepare__(meta, name, bases):
        return OrderedDict()

    @staticmethod
    def __new__(meta, clsname, bases, attrs):
        # Make named regex groups for all strings in the class body
        patterns = [ '(?P<%s>%s)' % (key, val) for key, val in attrs
.items()
                     if isinstance(val, str) ]

        # Make the master regex pattern
        attrs['_pattern'] = re.compile('|'.join(patterns))
        return super().__new__(meta, clsname, bases, attrs)
```

This metaclass inspects the class body for strings, makes named regex groups out of them, and forms a master regular expression. The use of an `OrderedDict` is to capture definition order—something that matters for proper regular expression matching.

Now, define a base class with the general `tokenize()` method:

```
class Tokenizer(metaclass=tokenizemeta):
    def tokenize(self, text):
        index = 0
        while index < len(text):
            m = self._pattern.match(text, index)
            if m:
                tokname = m.lastgroup
                toktext = m.group()
                if not tokname.startswith('ignore'):
                    yield (tokname, toktext)
                index = m.end()
            else:
                raise SyntaxError('Bad character %r' % text[index])
```

Now why did we go through all of this trouble? It makes the specification of a tokenizer easy. Try this:

```
class Simple(Tokenizer):
    NAME = r'[a-zA-Z_][a-zA-Z0-9_]*'
    NUM = r'\d+'
    ASSIGN = r'='
    PLUS = r'\+'
    TIMES = r'\*'
    ignore = r'\s+'

# Use the tokenizer
text = 'a = 3 + 4 * 5'
tokenizer = Simple()
for tok in tokenizer.tokenize(text):
    print(tok)
```

That's pretty cool. Using metaclasses, you were able to make a little specification language for tokenizing. The user of the `Tokenizer` class just gives the token names and regular expressions. The metaclass machinery behind the scenes takes care of the rest.

Adding Class Dictionary Magic

You can do even more with your tokenizer class if you're willing to stretch the definition of a dictionary. Let's subclass `OrderedDict` and change assignment slightly so that it detects duplicates:

```
class TokDict(OrderedDict):
    def __setitem__(self, key, value):
        if key in self and isinstance(key, str):
            raise KeyError('Token %s already defined' % key)
        else:
            super().__setitem__(key, value)

class tokenizemeta(type):
    @classmethod
    def __prepare__(meta, name, bases):
        return TokDict()

    ...
```

In this new version, a specification with a duplicate pattern name creates an error:

```
class Simple(Tokenizer):
    NAME = r'[a-zA-Z_][a-zA-Z0-9_]*'
    NUM = r'\d+'
    ASSIGN = r'='
    PLUS = r'\+'
    TIMES = r'\*'
    NUM = r'\d+'
    ignore = r'\s+'

Traceback (most recent call last):
  ...
KeyError: 'Token NUM already defined'
```

You could stretch it a bit further, though. This version allows optional action methods to be defined for any of the tokens:

```
from collections import OrderedDict
import re

class TokDict(OrderedDict):
    def __init__(self):
        super().__init__()
        self.actions = {}

    def __setitem__(self, key, value):
        if key in self and isinstance(key, str):
            if callable(value):
                self.actions[key] = value
            else:
                raise KeyError('Token %s already defined' % key)
        else:
            super().__setitem__(key, value)

class tokenizemeta(type):
    @classmethod
    def __prepare__(meta, name, bases):
        return TokDict()

    @staticmethod
    def __new__(meta, clsname, bases, attrs):
        # Make named regex groups for all strings in the class body
        patterns = [ '(?P<%s>%s)' % (key, val) for key, val in
attrs.items() if isinstance(val, str) ]
        # Make the master regex pattern
        attrs['_pattern'] = re.compile('|'.join(patterns))

        # Record action functions (if any)
        attrs['_actions'] = attrs.actions

        return super().__new__(meta, clsname, bases, attrs)

class Tokenizer(metaclass=tokenizemeta):
    def tokenize(self, text):
        index = 0
        while index < len(text):
            m = self._pattern.match(text, index)
            if m:
                tokname = m.lastgroup
                toktext = m.group()
                if not tokname.startswith('ignore'):
                    if tokname in self._actions:
                        yield (tokname, self._actions[tokname](self,
toktext))
                    else:
                        yield (tokname, toktext)
                index = m.end()
            else:
                raise SyntaxError('Bad character %r' % text[index])
```

This last one might require a bit of study, but it allows you to write a tokenizer like this:

```
class Simple(Tokenizer):
    NAME = r'[a-zA-Z_][a-zA-Z0-9_]*'
    NUM = r'\d+'
    ASSIGN = r'='
    PLUS = r'\+'
    TIMES = r'\*'
    ignore = r'\s+'
```

```
        # Convert NUM tokens to ints
        def NUM(self, text):
            return int(text)

        # Uppercase all names (case-insensitivity)
        def NAME(self, text):
            return text.upper()

    # Example
    text = 'a = 3 + 4 * 5'
    tokenizer = Simple()
    for tok in tokenizer.tokenize(text):
        print(tok)
```

If it's working, the final output should appear like this:

```
    ('NAME', 'A')
    ('ASSIGN', '=')
    ('NUM', 3)
    ('PLUS', '+')
    ('NUM', 4)
    ('TIMES', '*')
    ('NUM', 5)
```

Notice how the names have been uppercased and numbers converted to integers.

The Big Picture

By now, you're either staring at amazement or in horror at what we've done. In the big picture, one of the great powers of metaclasses is that you can use them to turn class definitions into a kind of small domain-specific language (DSL). By doing this, you can often simplify the specification of complex problems. Tokenization is just one such example. However, it's motivated by a long history of DSLs being used for various facets of software development (e.g., lex, yacc, RPC, interface definition languages, database models, etc.).

If you've used more advanced libraries or frameworks, chances are you've encountered metaclasses without even knowing it. For example, if you've ever used the Django Web framework, you describe database models using classes like this [3]:

```
    from django.db import models

    class Musician(models.Model):
        first_name = models.CharField(max_length=50)
        last_name = models.CharField(max_length=50)
        instrument = models.CharField(max_length=100)

    class Album(models.Model):
        artist = models.ForeignKey(Musician, on_delete=models
    .CASCADE)
        name = models.CharField(max_length=100)
        release_date = models.DateField()
        num_stars = models.IntegerField()
```

This involves metaclasses. It might not be obvious, but there is a whole set of code sitting behind the models.Model base class that is watching definitions and using that information to carry out various magic behind the scenes. A benefit of using a metaclass is that it can make it much easier for an end user to write specifications. They can write simple definitions and not worry so much about what's happening behind the scenes.

A Contrarian View

A common complaint lodged against metaclasses is that they introduce too much implicit magic into your program—violating the "Explicit is better than implicit" rule from the Zen of Python. To be sure, you don't actually need to use a metaclass to solve the problem presented here. For example, we possibly could have written a Tokenizer with more explicit data structures using a class definition like this:

```
    class Simple(Tokenizer):
        tokens = [
            ('NAME', r'[a-zA-Z_][a-zA-Z0-9_]*'),
            ('NUM', r'\d+'),
            ('ASSIGN', r'='),
            ('PLUS', r'\+'),
            ('TIMES', '\*'),
            ('ignore', r'\s+')
        ]
        actions = {
            'NAME': lambda text: text.upper(),
            'NUM': lambda text: int(text)
        }
```

It's not much more code than the metaclass version, but it frankly forces me to squint my eyes more than usual. Of course, they also say that beauty is in the eye of the beholder. So your mileage might vary.

Final Words

In parting, be on the lookout for metaclass magic the next time you use an interesting library or framework—they're often out there hiding in plain sight. If you're writing your own code and faced with problems involving complex or domain-specific specifications, metaclasses can be a useful tool for simplifying it.

References

[1] enum module: https://docs.python.org/3/library/enum.html.

[2] Customizing class creation (official documentation): https://docs.python.org/3/reference/datamodel.html#customizing-class-creation.

[3] Django models: https://docs.djangoproject.com/en/1.10/topics/db/models/.