CHAPTER 21
Class Metaprogramming

> [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).[1]
>
> — Tim Peters
> *Inventor of the timsort algorithm and prolific Python contributor*

Class metaprogramming is the art of creating or customizing classes at runtime. Classes are first-class objects in Python, so a function can be used to create a new class at any time, without using the `class` keyword. Class decorators are also functions, but capable of inspecting, changing, and even replacing the decorated class with another class. Finally, metaclasses are the most advanced tool for class metaprogramming: they let you create whole new categories of classes with special traits, such as the abstract base classes we've already seen.

Metaclasses are powerful, but hard to get right. Class decorators solve many of the same problems more simply. In fact, metaclasses are now so hard to justify in real code that my favorite motivating example lost much of its appeal with the introduction of class decorators in Python 2.6.

Also covered here is the distinction between import time and runtime: a crucial prerequisite for effective Python metaprogramming.

This is an exciting topic, and it's easy to get carried away. So I must start this chapter with the following admonition:

If you are not authoring a framework, you should not be writing metaclasses—unless you're doing it for fun or to practice the concepts.

1. Message to comp.lang.python, subject: "Acrimony in c.l.p.". This is another part of the same message from December 23, 2002, quoted in the Preface. The TimBot was inspired that day.

We'll get started by reviewing how to create a class at runtime.

A Class Factory

The standard library has a class factory that we've seen several times in this book: `collections.namedtuple`. It's a function that, given a class name and attribute names creates a subclass of `tuple` that allows retrieving items by name and provides a nice `__repr__` for debugging.

Sometimes I've felt the need for a similar factory for mutable objects. Suppose I'm writing a pet shop application and I want to process data for dogs as simple records. It's bad to have to write boilerplate like this:

```
class Dog:
    def __init__(self, name, weight, owner):
        self.name = name
        self.weight = weight
        self.owner = owner
```

Boring… the field names appear three times each. All that boilerplate doesn't even buy us a nice `repr`:

```
>>> rex = Dog('Rex', 30, 'Bob')
>>> rex
<__main__.Dog object at 0x2865bac>
```

Taking a hint from `collections.namedtuple`, let's create a `record_factory` that creates simple classes like `Dog` on the fly. Example 21-1 shows how it should work.

Example 21-1. Testing record_factory, a simple class factory

```
>>> Dog = record_factory('Dog', 'name weight owner')   ❶
>>> rex = Dog('Rex', 30, 'Bob')
>>> rex   ❷
Dog(name='Rex', weight=30, owner='Bob')
>>> name, weight, _ = rex   ❸
>>> name, weight
('Rex', 30)
>>> "{2}'s dog weighs {1}kg".format(*rex)   ❹
"Bob's dog weighs 30kg"
>>> rex.weight = 32   ❺
>>> rex
Dog(name='Rex', weight=32, owner='Bob')
>>> Dog.__mro__   ❻
(<class 'factories.Dog'>, <class 'object'>)
```

❶ Factory signature is similar to that of `namedtuple`: class name, followed by attribute names in a single string, separated by spaces or commas.

❷ Nice `repr`.

❸ Instances are iterable, so they can be conveniently unpacked on assignment…

❹ …or when passing to functions like `format`.

❺ A record instance is mutable.

❻ The newly created class inherits from `object`—no relationship to our factory.

The code for `record_factory` is in Example 21-2.[2]

Example 21-2. record_factory.py: a simple class factory

```
def record_factory(cls_name, field_names):
    try:
        field_names = field_names.replace(',', ' ').split()    ❶
    except AttributeError:  # no .replace or .split
        pass  # assume it's already a sequence of identifiers
    field_names = tuple(field_names)    ❷

    def __init__(self, *args, **kwargs):    ❸
        attrs = dict(zip(self.__slots__, args))
        attrs.update(kwargs)
        for name, value in attrs.items():
            setattr(self, name, value)

    def __iter__(self):    ❹
        for name in self.__slots__:
            yield getattr(self, name)

    def __repr__(self):    ❺
        values = ', '.join('{}={!r}'.format(*i) for i
                           in zip(self.__slots__, self))
        return '{}({})'.format(self.__class__.__name__, values)

    cls_attrs = dict(__slots__ = field_names,    ❻
                     __init__  = __init__,
                     __iter__  = __iter__,
                     __repr__  = __repr__)

    return type(cls_name, (object,), cls_attrs)    ❼
```

❶ Duck typing in practice: try to split `field_names` by commas or spaces; if that fails, assume it's already an iterable, with one name per item.

❷ Build a tuple of attribute names, this will be the `__slots__` attribute of the new class; this also sets the order of the fields for unpacking and `__repr__`.

❸ This function will become the `__init__` method in the new class. It accepts positional and/or keyword arguments.

2. Thanks to my friend J.S. Bueno for suggesting this solution.

❹ Implement an `__iter__`, so the class instances will be iterable; yield the field values in the order given by `__slots__`.

❺ Produce the nice `repr`, iterating over `__slots__` and `self`.

❻ Assemble dictionary of class attributes.

❼ Build and return the new class, calling the `type` constructor.

We usually think of `type` as a function, because we use it like one, e.g., `type(my_object)` to get the class of the object—same as `my_object.__class__`. However, `type` is a class. It behaves like a class that creates a new class when invoked with three arguments:

```
MyClass = type('MyClass', (MySuperClass, MyMixin),
               {'x': 42, 'x2': lambda self: self.x * 2})
```

The three arguments of `type` are named `name`, `bases`, and `dict`—the latter being a mapping of attribute names and attributes for the new class. The preceding code is functionally equivalent to this:

```
class MyClass(MySuperClass, MyMixin):
    x = 42

    def x2(self):
        return self.x * 2
```

The novelty here is that the instances of `type` are classes, like `MyClass` here, or the `Dog` class in Example 21-1.

In summary, the last line of `record_factory` in Example 21-2 builds a class named by the value of `cls_name`, with `object` as its single immediate superclass and with class attributes named `__slots__`, `__init__`, `__iter__`, and `__repr__`, of which the last three are instance methods.

We could have named the `__slots__` class attribute anything else, but then we'd have to implement `__setattr__` to validate the names of attributes being assigned, because for our record-like classes we want the set of attributes to be always the same and in the same order. However, recall that the main feature of `__slots__` is saving memory when you are dealing with millions of instances, and using `__slots__` has some drawbacks, discussed in "Saving Space with the __slots__ Class Attribute" on page 264.

Invoking `type` with three arguments is a common way of creating a class dynamically. If you peek at the source code for `collections.namedtuple`, you'll see a different approach: there is `_class_template`, a source code template as a string, and the `namedtuple` function fills its blanks calling `_class_template.format(…)`. The resulting source code string is then evaluated with the `exec` built-in function.

It's good practice to avoid `exec` or `eval` for metaprogramming in Python. These functions pose serious security risks if they are fed strings (even fragments) from untrusted sources. Python offers sufficient introspection tools to make `exec` and `eval` unnecessary most of the time. However, the Python core developers chose to use `exec` when implementing `namedtuple`. The chosen approach makes the code generated for the class available in the `._source` attribute.

Instances of classes created by `record_factory` have a limitation: they are not serializable—that is, they can't be used with the `dump/load` functions from the `pickle` module. Solving this problem is beyond the scope of this example, which aims to show the `type` class in action in a simple use case. For the full solution, study the source code for `collections.namedtuple`; search for the word "pickling."

A Class Decorator for Customizing Descriptors

When we left the `LineItem` example in "LineItem Take #5: A New Descriptor Type" on page 637, the issue of descriptive storage names was still pending: the value of attributes such as `weight` was stored in an instance attribute named `_Quantity#0`, which made debugging a bit hard. You can retrieve the storage name from a descriptor in Example 20-7 with the following lines:

```
>>> LineItem.weight.storage_name
'_Quantity#0'
```

However, it would be better if the storage names actually included the name of the managed attribute, like this:

```
>>> LineItem.weight.storage_name
'_Quantity#weight'
```

Recall from "LineItem Take #4: Automatic Storage Attribute Names" on page 631 that we could not use descriptive storage names because when the descriptor is instantiated it has no way of knowing the name of the managed attribute (i.e., the class attribute to which the descriptor will be bound, such as `weight` in the preceding examples). But once the whole class is assembled and the descriptors are bound to the class attributes, we can inspect the class and set proper storage names to the descriptors. This could be done in the `__new__` method of the `LineItem` class, so that by the time the descriptors are used in the `__init__` method, the correct storage names are set. The problem of using `__new__` for that purpose is wasted effort: the logic of `__new__` will run every time a new `LineItem` instance is created, but the binding of the descriptor to the managed attribute will never change once the `LineItem` class itself is built. So we need to set the

storage names when the class is created. That can be done with a class decorator or a metaclass. We'll do it first in the easier way.

A class decorator is very similar to a function decorator: it's a function that gets a class object and returns the same class or a modified one.

In Example 21-3, the `LineItem` class will be evaluated by the interpreter and the resulting class object will be passed to the `model.entity` function. Python will bind the global name `LineItem` to whatever the `model.entity` function returns. In this example, `model.entity` returns the same `LineItem` class with the `storage_name` attribute of each descriptor instance changed.

Example 21-3. bulkfood_v6.py: LineItem using Quantity and NonBlank descriptors

```
import model_v6 as model

@model.entity  ❶
class LineItem:
    description = model.NonBlank()
    weight = model.Quantity()
    price = model.Quantity()

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price
```

❶ The only change in this class is the added decorator.

Example 21-4 shows the implementation of the decorator. Only the new code at the bottom of *model_v6.py* is listed here; the rest of the module is identical to *model_v5.py* (Example 20-6).

Example 21-4. model_v6.py: a class decorator

```
def entity(cls):  ❶
    for key, attr in cls.__dict__.items():  ❷
        if isinstance(attr, Validated):  ❸
            type_name = type(attr).__name__
            attr.storage_name = '_{}#{}'.format(type_name, key)  ❹
    return cls  ❺
```

❶ Decorator gets class as argument.

❷ Iterate over `dict` holding the class attributes.

❸ If the attribute is one of our `Validated` descriptors…

❹ …set the `storage_name` to use the descriptor class name and the managed attribute name (e.g., `_NonBlank#description`).

❺ Return the modified class.

The doctests in *bulkfood_v6.py* prove that the changes are successful. For example, Example 21-5 shows the names of the storage attributes in a `LineItem` instance.

Example 21-5. bulkfood_v6.py: doctests for new storage_name descriptor attributes

```
>>> raisins = LineItem('Golden raisins', 10, 6.95)
>>> dir(raisins)[:3]
['_NonBlank#description', '_Quantity#price', '_Quantity#weight']
>>> LineItem.description.storage_name
'_NonBlank#description'
>>> raisins.description
'Golden raisins'
>>> getattr(raisins, '_NonBlank#description')
'Golden raisins'
```

That's not too complicated. Class decorators are a simpler way of doing something that previously required a metaclass: customizing a class the moment it's created.

A significant drawback of class decorators is that they act only on the class where they are directly applied. This means subclasses of the decorated class may or may not inherit the changes made by the decorator, depending on what those changes are. We'll explore the problem and see how it's solved in the following sections.

What Happens When: Import Time Versus Runtime

For successful metaprogramming, you must be aware of when the Python interpreter evaluates each block of code. Python programmers talk about "import time" versus "runtime" but the terms are not strictly defined and there is a gray area between them. At import time, the interpreter parses the source code of a *.py* module in one pass from top to bottom, and generates the bytecode to be executed. That's when syntax errors may occur. If there is an up-to-date *.pyc* file available in the local `__pycache__`, those steps are skipped because the bytecode is ready to run.

Although compiling is definitely an import-time activity, other things may happen at that time, because almost every statement in Python is executable in the sense that they potentially run user code and change the state of the user program. In particular, the `import` statement is not merely a declaration[3] but it actually runs all the top-level code of the imported module when it's imported for the first time in the process—further imports of the same module will use a cache, and only name binding occurs then. That

3. Contrast with the `import` statement in Java, which is just a declaration to let the compiler know that certain packages are required.

top-level code may do anything, including actions typical of "runtime", such as connecting to a database.[4] That's why the border between "import time" and "runtime" is fuzzy: the `import` statement can trigger all sorts of "runtime" behavior.

In the previous paragraph, I wrote that importing "runs all the top-level code," but "top-level code" requires some elaboration. The interpreter executes a `def` statement on the top level of a module when the module is imported, but what does that achieve? The interpreter compiles the function body (if it's the first time that module is imported), and binds the function object to its global name, but it does not execute the body of the function, obviously. In the usual case, this means that the interpreter defines top-level functions at import time, but executes their bodies only when—and if—the functions are invoked at runtime.

For classes, the story is different: at import time, the interpreter executes the body of every class, even the body of classes nested in other classes. Execution of a class body means that the attributes and methods of the class are defined, and then the class object itself is built. In this sense, the body of classes is "top-level code": it runs at import time.

This is all rather subtle and abstract, so here is an exercise to help you see what happens when.

The Evaluation Time Exercises

Consider a script, *evaltime.py*, which imports a module *evalsupport.py*. Both modules have several `print` calls to output markers in the format `<[N]>`, where `N` is a number. The goal of this pair of exercises is to determine when each of theses calls will be made.

Students have reported these exercises are helpful to better appreciate how Python evaluates the source code. Do take the time to solve them with paper and pencil before looking at "Solution for scenario #1" on page 664.

The listings are Examples 21-6 and 21-7. Grab paper and pencil and—without running the code—write down the markers in the order they will appear in the output, in two scenarios:

Scenario #1

The module *evaltime.py* is imported interactively in the Python console:

```
>>> import evaltime
```

4. I'm not saying starting a database connection just because a module is imported is a good idea, only pointing out it can be done.

Scenario #2

The module *evaltime.py* is run from the command shell:

```
$ python3 evaltime.py
```

Example 21-6. evaltime.py: write down the numbered <[N]> markers in the order they will appear in the output

```
from evalsupport import deco_alpha

print('<[1]> evaltime module start')


class ClassOne():
    print('<[2]> ClassOne body')

    def __init__(self):
        print('<[3]> ClassOne.__init__')

    def __del__(self):
        print('<[4]> ClassOne.__del__')

    def method_x(self):
        print('<[5]> ClassOne.method_x')

    class ClassTwo(object):
        print('<[6]> ClassTwo body')


@deco_alpha
class ClassThree():
    print('<[7]> ClassThree body')

    def method_y(self):
        print('<[8]> ClassThree.method_y')


class ClassFour(ClassThree):
    print('<[9]> ClassFour body')

    def method_y(self):
        print('<[10]> ClassFour.method_y')


if __name__ == '__main__':
    print('<[11]> ClassOne tests', 30 * '.')
    one = ClassOne()
    one.method_x()
    print('<[12]> ClassThree tests', 30 * '.')
    three = ClassThree()
    three.method_y()
    print('<[13]> ClassFour tests', 30 * '.')
    four = ClassFour()
```

```
    four.method_y()


print('<[14]> evaltime module end')
```

Example 21-7. evalsupport.py: module imported by evaltime.py

```
print('<[100]> evalsupport module start')

def deco_alpha(cls):
    print('<[200]> deco_alpha')

    def inner_1(self):
        print('<[300]> deco_alpha:inner_1')

    cls.method_y = inner_1
    return cls

# BEGIN META_ALEPH
class MetaAleph(type):
    print('<[400]> MetaAleph body')

    def __init__(cls, name, bases, dic):
        print('<[500]> MetaAleph.__init__')

        def inner_2(self):
            print('<[600]> MetaAleph.__init__:inner_2')

        cls.method_z = inner_2
# END META_ALEPH

print('<[700]> evalsupport module end')
```

Solution for scenario #1

Example 21-8 is the output of importing the *evaltime.py* module in the Python console.

Example 21-8. Scenario #1: importing evaltime in the Python console

```
>>> import evaltime
<[100]> evalsupport module start  ❶
<[400]> MetaAleph body  ❷
<[700]> evalsupport module end
<[1]> evaltime module start
<[2]> ClassOne body  ❸
<[6]> ClassTwo body  ❹
<[7]> ClassThree body
<[200]> deco_alpha  ❺
<[9]> ClassFour body
<[14]> evaltime module end  ❻
```

❶ All top-level code in `evalsupport` runs when the module is imported; the `deco_alpha` function is compiled, but its body does not execute.

❷ The body of the `MetaAleph` function does run.

❸ The body of every class is executed…

❹ …including nested classes.

❺ The decorator function runs after the body of the decorated `ClassThree` is evaluated.

❻ In this scenario, the `evaltime` is imported, so the `if __name__ == '__main__':` block never runs.

Notes about scenario #1:

1. This scenario is triggered by a simple `import evaltime` statement.
2. The interpreter executes every class body of the imported module and its dependency, `evalsupport`.
3. It makes sense that the interpreter evaluates the body of a decorated class before it invokes the decorator function that is attached on top of it: the decorator must get a class object to process, so the class object must be built first.
4. The only user-defined function or method that runs in this scenario is the `deco_alpha` decorator.

Now let's see what happens in scenario #2.

Solution for scenario #2

Example 21-9 is the output of running `python evaltime.py`.

Example 21-9. Scenario #2: running evaltime.py from the shell

```
$ python3 evaltime.py
<[100]> evalsupport module start
<[400]> MetaAleph body
<[700]> evalsupport module end
<[1]> evaltime module start
<[2]> ClassOne body
<[6]> ClassTwo body
<[7]> ClassThree body
<[200]> deco_alpha
<[9]> ClassFour body  ❶
<[11]> ClassOne tests .............................
<[3]> ClassOne.__init__  ❷
<[5]> ClassOne.method_x
<[12]> ClassThree tests ............................
<[300]> deco_alpha:inner_1  ❸
```

```
<[13]> ClassFour tests ..........................
<[10]> ClassFour.method_y
<[14]> evaltime module end
<[4]> ClassOne.__del__  ❹
```

❶ Same output as Example 21-8 so far.

❷ Standard behavior of a class.

❸ `ClassThree.method_y` was changed by the `deco_alpha` decorator, so the call `three.method_y()` runs the body of the `inner_1` function.

❹ The `ClassOne` instance bound to `one` global variable is garbage-collected only when the program ends.

The main point of scenario #2 is to show that the effects of a class decorator may not affect subclasses. In Example 21-6, `ClassFour` is defined as a subclass of `ClassThree`. The `@deco_alpha` decorator is applied to `ClassThree`, replacing its `method_y`, but that does not affect `ClassFour` at all. Of course, if the `ClassFour.method_y` did invoke the `ClassThree.method_y` with `super(…)`, we would see the effect of the decorator, as the `inner_1` function executed.

In contrast, the next section will show that metaclasses are more effective when we want to customize a whole class hierarchy, and not one class at a time.

Metaclasses 101

A metaclass is a class factory, except that instead of a function, like `record_factory` from Example 21-2, a metaclass is written as a class. Figure 21-1 depicts a metaclass using the Mills & Gizmos Notation: a mill producing another mill.

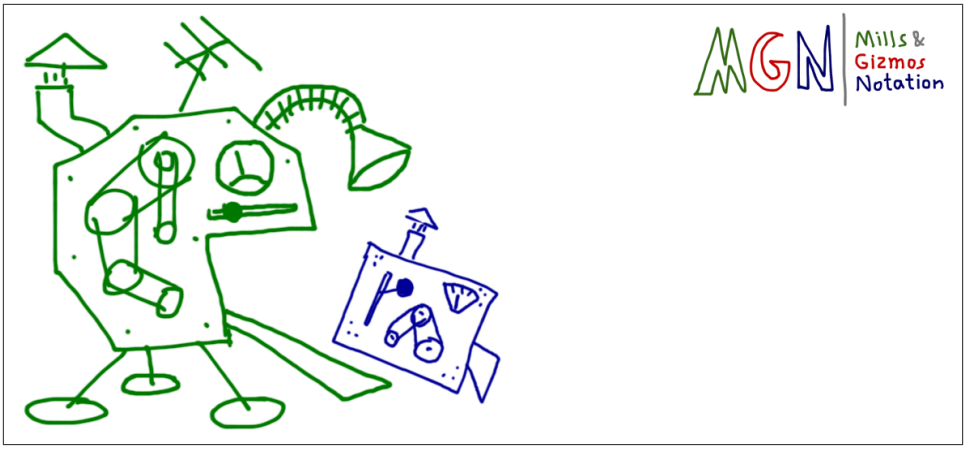

Figure 21-1. A metaclass is a class that builds classes

Consider the Python object model: classes are objects, therefore each class must be an instance of some other class. By default, Python classes are instances of `type`. In other words, `type` is the metaclass for most built-in and user-defined classes:

```
>>> 'spam'.__class__
<class 'str'>
>>> str.__class__
<class 'type'>
>>> from bulkfood_v6 import LineItem
>>> LineItem.__class__
<class 'type'>
>>> type.__class__
<class 'type'>
```

To avoid infinite regress, `type` is an instance of itself, as the last line shows.

Note that I am not saying that `str` or `LineItem` inherit from `type`. What I am saying is that `str` and `LineItem` are instances of `type`. They all are subclasses of `object`. Figure 21-2 may help you confront this strange reality.

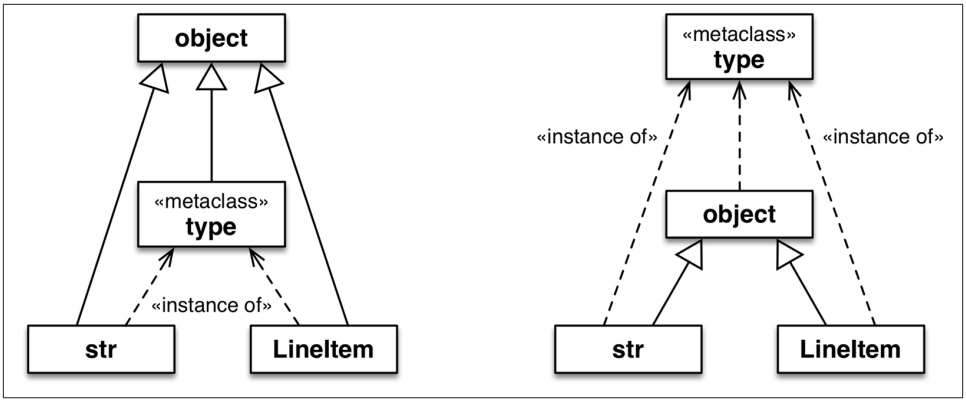

Figure 21-2. Both diagrams are true. The left one emphasizes that str, type, and LineItem are subclasses of object. The right one makes it clear that str, object, and LineItem are instances of type, because they are all classes.

The classes `object` and `type` have a unique relationship: `object` is an instance of `type`, and `type` is a subclass of `object`. This relationship is "magic": it cannot be expressed in Python because either class would have to exist before the other could be defined. The fact that `type` is an instance of itself is also magical.

Besides `type`, a few other metaclasses exist in the standard library, such as `ABCMeta` and `Enum`. The next snippet shows that the class of `collections.Iterable` is `abc.ABCMeta`. The class `Iterable` is abstract, but `ABCMeta` is not—after all, `Iterable` is an instance of `ABCMeta`:

```
>>> import collections
>>> collections.Iterable.__class__
<class 'abc.ABCMeta'>
>>> import abc
>>> abc.ABCMeta.__class__
<class 'type'>
>>> abc.ABCMeta.__mro__
(<class 'abc.ABCMeta'>, <class 'type'>, <class 'object'>)
```

Ultimately, the class of `ABCMeta` is also `type`. Every class is an instance of `type`, directly or indirectly, but only metaclasses are also subclasses of `type`. That's the most important relationship to understand metaclasses: a metaclass, such as `ABCMeta`, inherits from `type` the power to construct classes. Figure 21-3 illustrates this crucial relationship.

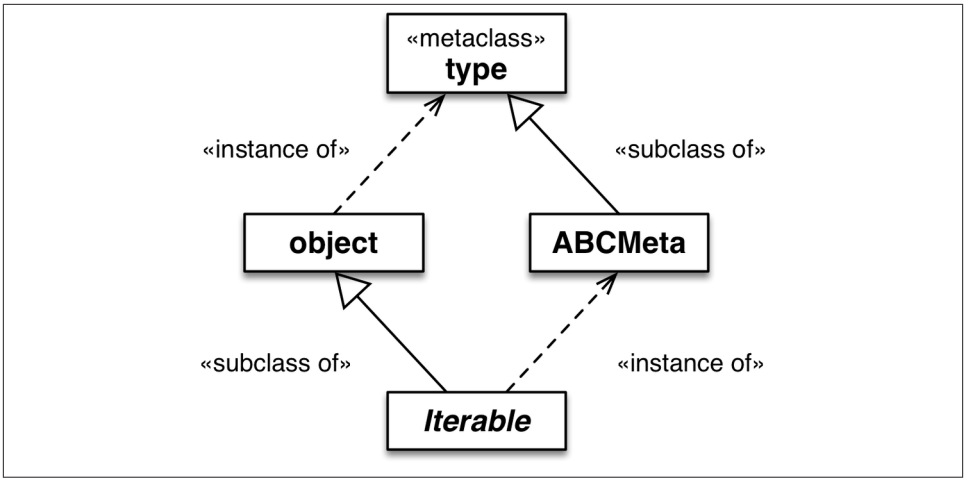

Figure 21-3. Iterable is a subclass of object and an instance of ABCMeta. Both object and ABCMeta are instances of type, but the key relationship here is that ABCMeta is also a subclass of type, because ABCMeta is a metaclass. In this diagram, Iterable is the only abstract class.

The important takeaway here is that all classes are instances of `type`, but metaclasses are also subclasses of `type`, so they act as class factories. In particular, a metaclass can customize its instances by implementing `__init__`. A metaclass `__init__` method can do everything a class decorator can do, but its effects are more profound, as the next exercise demonstrates.

The Metaclass Evaluation Time Exercise

This is a variation of "The Evaluation Time Exercises" on page 662. The *evalsupport.py* module is the same as Example 21-7, but the main script is now *evaltime_meta.py*, listed in Example 21-10.

Example 21-10. evaltime_meta.py: ClassFive is an instance of the MetaAleph metaclass

```
from evalsupport import deco_alpha
from evalsupport import MetaAleph

print('<[1]> evaltime_meta module start')


@deco_alpha
class ClassThree():
    print('<[2]> ClassThree body')

    def method_y(self):
        print('<[3]> ClassThree.method_y')
```

```
class ClassFour(ClassThree):
    print('<[4]> ClassFour body')

    def method_y(self):
        print('<[5]> ClassFour.method_y')


class ClassFive(metaclass=MetaAleph):
    print('<[6]> ClassFive body')

    def __init__(self):
        print('<[7]> ClassFive.__init__')

    def method_z(self):
        print('<[8]> ClassFive.method_y')


class ClassSix(ClassFive):
    print('<[9]> ClassSix body')

    def method_z(self):
        print('<[10]> ClassSix.method_y')


if __name__ == '__main__':
    print('<[11]> ClassThree tests', 30 * '.')
    three = ClassThree()
    three.method_y()
    print('<[12]> ClassFour tests', 30 * '.')
    four = ClassFour()
    four.method_y()
    print('<[13]> ClassFive tests', 30 * '.')
    five = ClassFive()
    five.method_z()
    print('<[14]> ClassSix tests', 30 * '.')
    six = ClassSix()
    six.method_z()

print('<[15]> evaltime_meta module end')
```

Again, grab pencil and paper and write down the numbered `<[N]>` markers in the order they will appear in the output, considering these two scenarios:

Scenario #3
: The module *evaltime_meta.py* is imported interactively in the Python console.

Scenario #4
: The module *evaltime_meta.py* is run from the command shell.

Solutions and analysis are next.

Solution for scenario #3

Example 21-11 shows the output of importing *evaltime_meta.py* in the Python console.

Example 21-11. Scenario #3: importing evaltime_meta in the Python console

```
>>> import evaltime_meta
<[100]> evalsupport module start
<[400]> MetaAleph body
<[700]> evalsupport module end
<[1]> evaltime_meta module start
<[2]> ClassThree body
<[200]> deco_alpha
<[4]> ClassFour body
<[6]> ClassFive body
<[500]> MetaAleph.__init__  ❶
<[9]> ClassSix body
<[500]> MetaAleph.__init__  ❷
<[15]> evaltime_meta module end
```

❶ The key difference from scenario #1 is that the `MetaAleph.__init__` method is invoked to initialize the just-created `ClassFive`.

❷ And `MetaAleph.__init__` also initializes `ClassSix`, which is a subclass of `ClassFive`.

The Python interpreter evaluates the body of `ClassFive` but then, instead of calling `type` to build the actual class body, it calls `MetaAleph`. Looking at the definition of `MetaAleph` in Example 21-12, you'll see that the `__init__` method gets four arguments:

`self`
: That's the class object being initialized (e.g., `ClassFive`)

`name`, `bases`, `dic`
: The same arguments passed to `type` to build a class

Example 21-12. evalsupport.py: definition of the metaclass MetaAleph from Example 21-7

```
class MetaAleph(type):
    print('<[400]> MetaAleph body')

    def __init__(cls, name, bases, dic):
        print('<[500]> MetaAleph.__init__')

        def inner_2(self):
            print('<[600]> MetaAleph.__init__:inner_2')

        cls.method_z = inner_2
```

When coding a metaclass, it's conventional to replace `self` with `cls`. For example, in the `__init__` method of the metaclass, using `cls` as the name of the first argument makes it clear that the instance under construction is a class.

The body of `__init__` defines an `inner_2` function, then binds it to `cls.method_z`. The name `cls` in the signature of `MetaAleph.__init__` refers to the class being created (e.g., `ClassFive`). On the other hand, the name `self` in the signature of `inner_2` will eventually refer to an instance of the class we are creating (e.g., an instance of `ClassFive`).

Solution for scenario #4

Example 21-13 shows the output of running `python evaltime.py` from the command line.

Example 21-13. Scenario #4: running evaltime_meta.py from the shell

```
$ python3 evaltime.py
<[100]> evalsupport module start
<[400]> MetaAleph body
<[700]> evalsupport module end
<[1]> evaltime_meta module start
<[2]> ClassThree body
<[200]> deco_alpha
<[4]> ClassFour body
<[6]> ClassFive body
<[500]> MetaAleph.__init__
<[9]> ClassSix body
<[500]> MetaAleph.__init__
<[11]> ClassThree tests ..............................
<[300]> deco_alpha:inner_1  ❶
<[12]> ClassFour tests ..............................
<[5]> ClassFour.method_y  ❷
<[13]> ClassFive tests ..............................
<[7]> ClassFive.__init__
<[600]> MetaAleph.__init__:inner_2  ❸
<[14]> ClassSix tests ..............................
<[7]> ClassFive.__init__
<[600]> MetaAleph.__init__:inner_2  ❹
<[15]> evaltime_meta module end
```

❶ When the decorator is applied to `ClassThree`, its `method_y` is replaced by the `inner_1` method…

❷ But this has no effect on the undecorated `ClassFour`, even though `ClassFour` is a subclass of `ClassThree`.

❸ The `__init__` method of `MetaAleph` replaces `ClassFive.method_z` with its `inner_2` function.

❹ The same happens with the `ClassFive` subclass, `ClassSix`: its `method_z` is replaced by `inner_2`.

Note that `ClassSix` makes no direct reference to `MetaAleph`, but it is affected by it because it's a subclass of `ClassFive` and therefore it is also an instance of `MetaAleph`, so it's initialized by `MetaAleph.__init__`.

Further class customization can be done by implementing `__new__` in a metaclass. But more often than not, implementing `__init__` is enough.

We can now put all this theory in practice by creating a metaclass to provide a definitive solution to the descriptors with automatic storage attribute names.

A Metaclass for Customizing Descriptors

Back to the `LineItem` examples. It would be nice if the user did not have to be aware of decorators or metaclasses at all, and could just inherit from a class provided by our library, like in Example 21-14.

Example 21-14. bulkfood_v7.py: inheriting from model.Entity can work, if a metaclass is behind the scenes

```
import model_v7 as model

class LineItem(model.Entity):    ❶
    description = model.NonBlank()
    weight = model.Quantity()
    price = model.Quantity()

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price
```

❶ `LineItem` is a subclass of `model.Entity`.

Example 21-14 looks pretty harmless. No strange syntax to be seen at all. However, it only works because *model_v7.py* defines a metaclass, and `model.Entity` is an instance of that metaclass. Example 21-15 shows the implementation of the `Entity` class in the *model_v7.py* module.

Example 21-15. model_v7.py: the EntityMeta metaclass and one instance of it, Entity

```
class EntityMeta(type):
    """Metaclass for business entities with validated fields"""

    def __init__(cls, name, bases, attr_dict):
        super().__init__(name, bases, attr_dict)   ❶
        for key, attr in attr_dict.items():   ❷
            if isinstance(attr, Validated):
                type_name = type(attr).__name__
                attr.storage_name = '_{}#{}'.format(type_name, key)

class Entity(metaclass=EntityMeta):   ❸
    """Business entity with validated fields"""
```

❶ Call `__init__` on the superclass (`type` in this case).

❷ Same logic as the `@entity` decorator in Example 21-4.

❸ This class exists for convenience only: the user of this module can just subclass `Entity` and not worry about `EntityMeta`—or even be aware of its existence.

The code in Example 21-14 passes the tests in Example 21-3. The support module, *model_v7.py*, is harder to understand than *model_v6.py*, but the user-level code is simpler: just inherit from `model_v7.entity` and you get custom storage names for your `Validated` fields.

Figure 21-4 is a simplified depiction of what we just implemented. There is a lot going on, but the complexity is hidden inside the `model_v7` module. From the user perspective, `LineItem` is simply a subclass of `Entity`, as coded in Example 21-14. This is the power of abstraction.

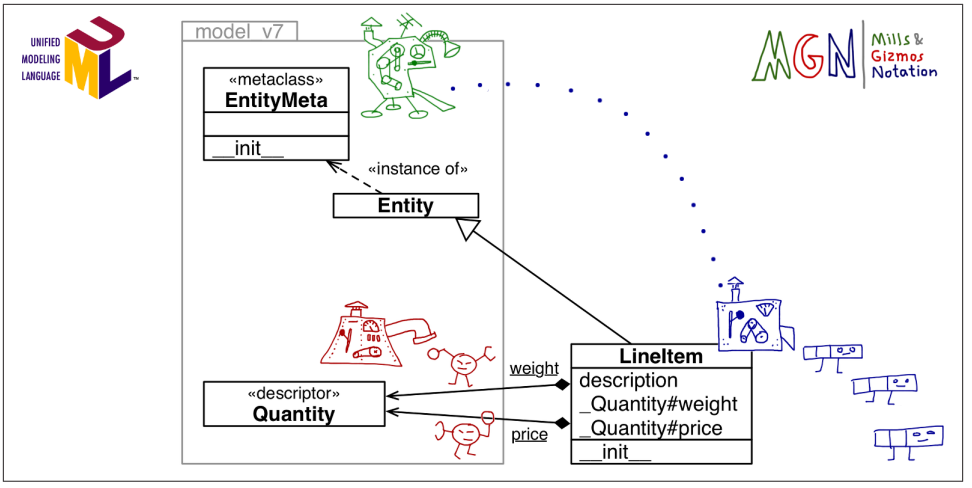

Figure 21-4. UML class diagram annotated with MGN (Mills & Gizmos Notation): the EntityMeta meta-mill builds the LineItem mill. Configuration of the descriptors (e.g., weight and price) is done by EntityMeta.__init__. Note the package boundary of model_v7.

Except for the syntax for linking a class to the metaclass,[5] everything written so far about metaclasses applies to versions of Python as early as 2.2, when Python types underwent a major overhaul. The next section covers a feature that is only available in Python 3.

The Metaclass ___prepare___ Special Method

In some applications it's interesting to be able to know the order in which the attributes of a class are defined. For example, a library to read/write CSV files driven by user-defined classes may want to map the order of the fields declared in the class to the order of the columns in the CSV file.

As we've seen, both the `type` constructor and the `__new__` and `__init__` methods of metaclasses receive the body of the class evaluated as a mapping of names to attributes. However, by default, that mapping is a `dict`, which means the order of the attributes as they appear in the class body is lost by the time our metaclass or class decorator can look at them.

The solution to this problem is the `__prepare__` special method, introduced in Python 3. This special method is relevant only in metaclasses, and it must be a class method (i.e., defined with the `@classmethod` decorator). The `__prepare__` method is invoked

5. Recall from "ABC Syntax Details" on page 328 that in Python 2.7 the `__metaclass__` class attribute is used, and the `metaclass=` keyword argument is not supported in the class declaration.

by the interpreter before the `__new__` method in the metaclass to create the mapping that will be filled with the attributes from the class body. Besides the metaclass as first argument, `__prepare__` gets the name of the class to be constructed and its tuple of base classes, and it must return a mapping, which will be received as the last argument by `__new__` and then `__init__` when the metaclass builds a new class.

It sounds complicated in theory, but in practice, every time I've seen `__prepare__` being used it was very simple. Take a look at Example 21-16.

Example 21-16. model_v8.py: the EntityMeta metaclass uses prepare, and Entity now has a field_names class method

```
class EntityMeta(type):
    """Metaclass for business entities with validated fields"""

    @classmethod
    def __prepare__(cls, name, bases):
        return collections.OrderedDict()  ❶

    def __init__(cls, name, bases, attr_dict):
        super().__init__(name, bases, attr_dict)
        cls._field_names = []  ❷
        for key, attr in attr_dict.items():  ❸
            if isinstance(attr, Validated):
                type_name = type(attr).__name__
                attr.storage_name = '_{}#{}'.format(type_name, key)
                cls._field_names.append(key)  ❹


class Entity(metaclass=EntityMeta):
    """Business entity with validated fields"""

    @classmethod
    def field_names(cls):  ❺
        for name in cls._field_names:
            yield name
```

❶ Return an empty `OrderedDict` instance, where the class attributes will be stored.

❷ Create a `_field_names` attribute in the class under construction.

❸ This line is unchanged from the previous version, but `attr_dict` here is the `OrderedDict` obtained by the interpreter when it called `__prepare__` before calling `__init__`. Therefore, this `for` loop will go over the attributes in the order they were added.

❹ Add the name of each `Validated` field found to `_field_names`.

❺ The `field_names` class method simply yields the names of the fields in the order they were added.

With the simple additions made in Example 21-16, we are now able to iterate over the `Validated` fields of any `Entity` subclass using the `field_names` class method. Example 21-17 demonstrates this new feature.

Example 21-17. bulkfood_v8.py: doctest showing the use of field_names—no changes are needed in the LineItem class; field_names is inherited from model.Entity

```
    >>> for name in LineItem.field_names():
    ...     print(name)
    ...
    description
    weight
    price
```

This wraps up our coverage of metaclasses. In the real world, metaclasses are used in frameworks and libraries that help programmers perform, among other tasks:

- Attribute validation
- Applying decorators to many methods at once
- Object serialization or data conversion
- Object-relational mapping
- Object-based persistency
- Dynamic translation of class structures from other languages

We'll now have a brief overview of methods defined in the Python data model for all classes.

Classes as Objects

Every class has a number of attributes defined in the Python data model, documented in "4.13. Special Attributes" of the "Built-in Types" chapter in the *Library Reference*. Three of those attributes we've seen several times in the book already: `__mro__`, `__class__`, and `__name__`. Other class attributes are:

`cls.__bases__`
: The tuple of base classes of the class.

`cls.__qualname__`
: A new attribute in Python 3.3 holding the qualified name of a class or function, which is a dotted path from the global scope of the module to the class definition. For example, in Example 21-6, the `__qualname__` of the inner class `ClassTwo` is the string `'ClassOne.ClassTwo'`, while its `__name__` is just `'ClassTwo'`. The specification for this attribute is PEP-3155 — Qualified name for classes and functions.

`cls.__subclasses__()`
: This method returns a list of the immediate subclasses of the class. The implementation uses weak references to avoid circular references between the superclass and its subclasses—which hold a strong reference to the superclasses in their `__bases__` attribute. The method returns the list of subclasses that currently exist in memory.

`cls.mro()`
: The interpreter calls this method when building a class to obtain the tuple of superclasses that is stored in the `__mro__` attribute of the class. A metaclass can override this method to customize the method resolution order of the class under construction.

None of the attributes mentioned in this section are listed by the `dir(…)` function.

With this, our study of class metaprogramming ends. This is a vast topic and I only scratched the surface. That's why we have "Further Reading" sections in this book.

Chapter Summary

Class metaprogramming is about creating or customizing classes dynamically. Classes in Python are first-class objects, so we started the chapter by showing how a class can be created by a function invoking the `type` built-in metaclass.

In the next section, we went back to the `LineItem` class with descriptors from Chapter 20 to solve a lingering issue: how to generate names for the storage attributes that reflected the names of the managed attributes (e.g., `_Quantity#price` instead of `_Quantity#1`). The solution was to use a class decorator, essentially a function that gets a just-built class and has the opportunity to inspect it, change it, and even replace it with a different class.

We then moved to a discussion of when different parts of the source code of a module actually run. We saw that there is some overlap between the so-called "import time" and "runtime," but clearly a lot of code runs triggered by the `import` statement. Understanding what runs when is crucial, and there are some subtle rules, so we used the evaluation-time exercises to cover this topic.

The following subject was an introduction to metaclasses. We saw that all classes are instances of `type`, directly or indirectly, so that is the "root metaclass" of the language. A variation of the evaluation-time exercise was designed to show that a metaclass can

customize a hierarchy of classes—in contrast with a class decorator, which affects a single class and may have no impact on its descendants.

The first practical application of a metaclass was to solve the issue of the storage attribute names in `LineItem`. The resulting code is a bit trickier than the class decorator solution, but it can be encapsulated in a module so that the user merely subclasses an apparently plain class (`model.Entity`) without being aware that it is an instance of a custom metaclass (`model.EntityMeta`). The end result is reminiscent of the ORM APIs in Django and SQLAlchemy, which use metaclasses in their implementations but don't require the user to know anything about them.

The second metaclass we implemented added a small feature to `model.EntityMeta`: a `__prepare__` method to provide an `OrderedDict` to serve as the mapping from names to attributes. This preserves the order in which those attributes are bound in the body of the class under construction, so that metaclass methods like `__new__` and `__init__` can use that information. In the example, we implemented a `_field_names` class attribute, which made possible an `Entity.field_names()` so users could retrieve the `Validated` descriptors in the same order they appear in the source code.

The last section was a brief overview of attributes and methods available in all Python classes.

Metaclasses are challenging, exciting, and—sometimes—abused by programmers trying to be too clever. To wrap up, let's recall Alex Martelli's final advice from his essay "Waterfowl and ABCs" on page 314:

> And, *don't* define custom ABCs (or metaclasses) in production code… if you feel the urge to do so, I'd bet it's likely to be a case of "all problems look like a nail"-syndrome for somebody who just got a shiny new hammer—you (and future maintainers of your code) will be much happier sticking with straightforward and simple code, eschewing such depths.
>
> — Alex Martelli

Wise words from a man who is not only a master of Python metaprogramming but also an accomplished software engineer working on some of the largest mission-critical Python deployments in the world.

Further Reading

The essential references for this chapter in the Python documentation are "3.3.3. Customizing class creation" in the "Data Model" chapter of The Python Language Reference, the `type` class documentation in the "Built-in Functions" page, and "4.13. Special Attributes" of the "Built-in Types" chapter in the *Library Reference*. Also, in the *Library Reference*, the `types` module documentation covers two functions that are new in

Python 3.3 and are designed to help with class metaprogramming: `types.new_class(…)` and `types.prepare_class(…)`.

Class decorators were formalized in PEP 3129 - Class Decorators, written by Collin Winter, with the reference implementation authored by Jack Diederich. The PyCon 2009 talk "Class Decorators: Radically Simple" (video), also by Jack Diederich, is a quick introduction to the feature.

Python in a Nutshell, 2E by Alex Martelli features outstanding coverage of metaclasses, including a `metaMetaBunch` metaclass that aims to solve the same problem as our simple `record_factory` from Example 21-2 but is much more sophisticated. Martelli does not address class decorators because the feature appeared later than his book. Beazley and Jones provide excellent examples of class decorators and metaclasses in their *Python Cookbook, 3E* (O'Reilly). Michael Foord wrote an intriguing post titled "Meta-classes Made Easy: Eliminating self with Metaclasses". The subtitle says it all.

For metaclasses, the main references are PEP 3115 — Metaclasses in Python 3000, in which the `__prepare__` special method was introduced and Unifying types and classes in Python 2.2, authored by Guido van Rossum. The text applies to Python 3 as well, and it covers what were then called the "new-style" class semantics, including descriptors and metaclasses. It's a must-read. One of the references cited by Guido is *Putting Metaclasses to Work: a New Dimension in Object-Oriented Programming*, by Ira R. Forman and Scott H. Danforth (Addison-Wesley, 1998), a book to which he gave 5 stars on Amazon.com, adding the following review:

> **This book contributed to the design for metaclasses in Python 2.2**
>
> Too bad this is out of print; I keep referring to it as the best tutorial I know for the difficult subject of cooperative multiple inheritance, supported by Python via the `super()` function.[6].

For Python 3.5—in alpha as I write this—PEP 487 - Simpler customization of class creation puts forward a new special method, `__init_subclass__` that will allow a regular class (i.e., not a metaclass) to customize the initialization of its subclasses. As with class decorators, `__init_subclass__` will make class metaprogramming more accessible and also make it that much harder to justify the deployment of the nuclear option —metaclasses.

If you are into metaprogramming, you may wish Python had the ultimate metaprogramming feature: syntactic macros, as offered by Elixir and the Lisp family of languages. Be careful what you wish for. I'll just say one word: MacroPy.

6. Amazon.com catalog page for *Putting Metaclasses to Work*. You can still buy it used. I bought it and found it a hard read, but I will probably go back to it later.

Soapbox

I will start the last soapbox in the book with a long quote from Brian Harvey and Matthew Wright, two computer science professors from the University of California (Berkeley and Santa Barbara). In their book, *Simply Scheme*, Harvey and Wright wrote:

> There are two schools of thought about teaching computer science. We might caricature the two views this way:
>
> 1. **The conservative view**: Computer programs have become too large and complex to encompass in a human mind. Therefore, the job of computer science education is to teach people how to discipline their work in such a way that 500 mediocre programmers can join together and produce a program that correctly meets its specification.
> 2. **The radical view**: Computer programs have become too large and complex to encompass in a human mind. Therefore, the job of computer science education is to teach people how to expand their minds so that the programs can fit, by learning to think in a vocabulary of larger, more powerful, more flexible ideas than the obvious ones. Each unit of programming thought must have a big payoff in the capabilities of the program.[7]
>
> — Brian Harvey and Matthew Wright
> *Preface to Simply Scheme*

Harvey and Wright's exaggerated descriptions are about teaching computer science, but they also apply to programming language design. By now, you should have guessed that I subscribe to the "radical" view, and I believe Python was designed in that spirit.

The property idea is a great step forward compared to the accessors-from-the-start approach practically demanded by Java and supported by Java IDEs generating getters/setters with a keyboard shortcut. The main advantage of properties is to let us start our programs simply exposing attributes as public—in the spirit of *KISS*—knowing a public attribute can become a property at any time without much pain. But the descriptor idea goes way beyond that, providing a framework for abstracting away repetitive accessor logic. That framework is so effective that essential Python constructs use it behind the scenes.

Another powerful idea is functions as first-class objects, paving the way to higher-order functions. Turns out the combination of descriptors and higher-order functions enable the unification of functions and methods. A function's `__get__` produces a method object on the fly by binding the instance to the `self` argument. This is elegant.[8]

7. Brian Harvey and Matthew Wright, *Simply Scheme* (MIT Press, 1999), p. xvii. Full text available at Berkeley.edu.
8. *Machine Beauty* by David Gelernter (Basic Books) is an intriguing short book about elegance and aesthetics in works of engineering, from bridges to software.

Finally, we have the idea of classes as first-class objects. It's an outstanding feat of design that a beginner-friendly language provides powerful abstractions such as class decorators and full-fledged, user-defined metaclasses. Best of all: the advanced features are integrated in a way that does not complicate Python's suitability for casual programming (they actually help it, under the covers). The convenience and success of frameworks such as Django and SQLAlchemy owes much to metaclasses, even if many users of these tools aren't aware of them. But they can always learn and create the next great library.

I haven't yet found a language that manages to be easy for beginners, practical for professionals, and exciting for hackers in the way that Python is. Thanks, Guido van Rossum and everybody else who makes it so.