-
- Python Cookbook

- Comments Off
- Chapters

Table of Contents

Chapter 8. Classes and Objects

The primary focus of this chapter is to present recipes to common programming patterns related to class definitions. Topics include making objects support common Python features, usage of special methods, encapsulation techniques, inheritance, memory management, and useful design patterns.

Changing the String Representation of Instances

Problem

You want to change the output produced by printing or viewing instances to something more sensible.

Solution

To change the string representation of an instance, define the `__str__()` and `__repr__()` methods. For example:

```
class Pair:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __repr__(self):
        return 'Pair({0.x!r}, {0.y!r})'.format(self)
    def __str__(self):
        return '({0.x!s}, {0.y!s})'.format(self)
```

The `__repr__()` method returns the code representation of an instance, and is usually the text you would type to re-create the instance. The built-in `repr()` function returns this text, as does the interactive interpreter when inspecting values. The `__str__()` method converts the instance to a string, and is the output produced by the `str()` and `print()` functions. For example:

```
>>> p = Pair(3, 4)
>>> p
Pair(3, 4)          # __repr__() output
>>> print(p)
(3, 4)              # __str__() output
>>>
```

The implementation of this recipe also shows how different string representations may be used during formatting. Specifically, the special `!r` formatting code indicates that the output of `__repr__()` should be used instead of `__str__()`, the default. You can try this experiment with the preceding class to see this:

```
>>> p = Pair(3, 4)
>>> print('p is {0!r}'.format(p))
p is Pair(3, 4)
>>> print('p is {0}'.format(p))
p is (3, 4)
>>>
```

Discussion

Defining `__repr__()` and `__str__()` is often good practice, as it can simplify debugging and instance output. For example, by merely printing or logging an instance, a programmer will be shown more useful information about the instance contents.

It is standard practice for the output of `__repr__()` to produce text such that `eval(repr(x)) == x`. If this is not possible or desired, then it is common to create a useful textual representation enclosed in < and > instead. For example:

```
>>> f = open('file.dat')
>>> f
<_io.TextIOWrapper name='file.dat' mode='r' encoding='UTF-8'>
>>>
```

If no `__str__()` is defined, the output of `__repr__()` is used as a fallback.

The use of `format()` in the solution might look a little funny, but the format code `{0.x}` specifies the `x`-attribute of argument 0. So, in the following function, the `0` is actually the instance `self`:

```
def __repr__(self):
    return 'Pair({0.x!r}, {0.y!r})'.format(self)
```

As an alternative to this implementation, you could also use the `%` operator and the following code:

```
def __repr__(self):
    return 'Pair(%r, %r)' % (self.x, self.y)
```

Customizing String Formatting

Problem

You want an object to support customized formatting through the `format()` function and string method.

Solution

To customize string formatting, define the `__format__()` method on a class. For example:

```
_formats = {
    'ymd' : '{d.year}-{d.month}-{d.day}',
    'mdy' : '{d.month}/{d.day}/{d.year}',
    'dmy' : '{d.day}/{d.month}/{d.year}'
    }

class Date:
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

    def __format__(self, code):
        if code == '':
            code = 'ymd'
        fmt = _formats[code]
        return fmt.format(d=self)
```

Instances of the `Date` class now support formatting operations such as the following:

```
>>> d = Date(2012, 12, 21)
>>> format(d)
'2012-12-21'
>>> format(d, 'mdy')
'12/21/2012'
>>> 'The date is {:ymd}'.format(d)
'The date is 2012-12-21'
>>> 'The date is {:mdy}'.format(d)
'The date is 12/21/2012'
>>>
```

Discussion

The `__format__()` method provides a hook into Python's string formatting functionality. It's important to emphasize that the interpretation of format codes is entirely up to the class itself. Thus, the codes can be almost anything at all. For example, consider the following from the `datetime` module:

```
>>> from datetime import date
>>> d = date(2012, 12, 21)
>>> format(d)
'2012-12-21'
>>> format(d,'%A, %B %d, %Y')
'Friday, December 21, 2012'
>>> 'The end is {:%d %b %Y}. Goodbye'.format(d)
'The end is 21 Dec 2012. Goodbye'
>>>
```

There are some standard conventions for the formatting of the built-in types. See the documentation for the `string` module for a formal specification.

Making Objects Support the Context-Management Protocol

Problem

You want to make your objects support the context-management protocol (the `with` statement).

Solution

In order to make an object compatible with the `with` statement, you need to implement `__enter__()` and `__exit__()` methods. For example, consider the following class, which provides a network connection:

```
from socket import socket, AF_INET, SOCK_STREAM

class LazyConnection:
    def __init__(self, address, family=AF_INET, type=SOCK_STREAM):
        self.address = address
        self.family = AF_INET
        self.type = SOCK_STREAM
        self.sock = None

    def __enter__(self):
        if self.sock is not None:
            raise RuntimeError('Already connected')
        self.sock = socket(self.family, self.type)
        self.sock.connect(self.address)
        return self.sock

    def __exit__(self, exc_ty, exc_val, tb):
        self.sock.close()
        self.sock = None
```

The key feature of this class is that it represents a network connection, but it doesn't actually do anything initially (e.g., it doesn't establish a connection). Instead, the connection is established and closed using the `with` statement (essentially on demand). For example:

```
from functools import partial

conn = LazyConnection(('www.python.org', 80))
# Connection closed
```

```
with conn as s:
    # conn.__enter__() executes: connection open
    s.send(b'GET /index.html HTTP/1.0\r\n')
    s.send(b'Host: www.python.org\r\n')
    s.send(b'\r\n')
    resp = b''.join(iter(partial(s.recv, 8192), b''))
    # conn.__exit__() executes: connection closed
```

Discussion

The main principle behind writing a context manager is that you're writing code that's meant to surround a block of statements as defined by the use of the `with` statement. When the `with` statement is first encountered, the `__enter__()` method is triggered. The return value of `__enter__()` (if any) is placed into the variable indicated with the `as` qualifier. Afterward, the statements in the body of the `with` statement execute. Finally, the `__exit__()` method is triggered to clean up.

This control flow happens regardless of what happens in the body of the `with` statement, including if there are exceptions. In fact, the three arguments to the `__exit__()` method contain the exception type, value, and traceback for pending exceptions (if any). The `__exit__()` method can choose to use the exception information in some way or to ignore it by doing nothing and returning `None` as a result. If `__exit__()` returns `True`, the exception is cleared as if nothing happened and the program continues executing statements immediately after the `with` block.

One subtle aspect of this recipe is whether or not the `LazyConnection` class allows nested use of the connection with multiple `with` statements. As shown, only a single socket connection at a time is allowed, and an exception is raised if a repeated `with` statement is attempted when a socket is already in use. You can work around this limitation with a slightly different implementation, as shown here:

```
from socket import socket, AF_INET, SOCK_STREAM

class LazyConnection:
    def __init__(self, address, family=AF_INET, type=SOCK_STREAM):
        self.address = address
        self.family = AF_INET
        self.type = SOCK_STREAM
        self.connections = []

    def __enter__(self):
        sock = socket(self.family, self.type)
        sock.connect(self.address)
        self.connections.append(sock)
        return sock

    def __exit__(self, exc_ty, exc_val, tb):
        self.connections.pop().close()

# Example use
from functools import partial

conn = LazyConnection(('www.python.org', 80))
with conn as s1:
     ...
     with conn as s2:
          ...
          # s1 and s2 are independent sockets
```

In this second version, the `LazyConnection` class serves as a kind of factory for connections. Internally, a list is used to keep a stack. Whenever `__enter__()` executes, it makes a new connection and adds it to the stack. The `__exit__()` method simply pops the last connection off the stack and closes it. It's subtle, but this allows multiple connections to be created at once with nested `with` statements, as shown.

Context managers are most commonly used in programs that need to manage resources such as files, network connections, and locks. A key part of such resources is they have to be explicitly closed or released to operate correctly. For instance, if you acquire a lock, then you have to make sure you release it, or else you risk deadlock. By implementing `__enter__()`, `__exit__()`, and using the `with` statement, it is much easier to avoid such problems, since the cleanup code in the `__exit__()` method is guaranteed to run no matter what.

An alternative formulation of context managers is found in the `contextmanager` module. See "Defining Context Managers the Easy Way". A thread-safe version of this recipe can be found in "Storing Thread-Specific State".

Saving Memory When Creating a Large Number of Instances

Problem

Your program creates a large number (e.g., millions) of instances and uses a large amount of memory.

Solution

For classes that primarily serve as simple data structures, you can often greatly reduce the memory footprint of instances by adding the `__slots__` attribute to the class definition. For example:

```
class Date:
    __slots__ = ['year', 'month', 'day']
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day
```

When you define `__slots__`, Python uses a much more compact internal representation for instances. Instead of each instance consisting of a dictionary, instances are built around a small fixed-sized array, much like a tuple or list. Attribute names listed in the `__slots__` specifier are internally mapped to specific indices within this array. A side effect of using slots is that it is no longer possible to add new attributes to instances—you are restricted to only those attribute names listed in the `__slots__` specifier.

Discussion

The memory saved by using slots varies according to the number and type of attributes stored. However, in general, the resulting memory use is comparable to that of storing data in a tuple. To give you an idea, storing a single `Date` instance without slots requires 428 bytes of memory on a 64-bit version of Python. If slots is defined, it drops to 156 bytes. In a program that manipulated a large number of dates all at once, this would make a significant reduction in overall memory use.

Although slots may seem like a feature that could be generally useful, you should resist the urge to use it in most code. There are many parts of Python that rely on the normal dictionary-based implementation. In addition, classes that define slots don't support certain features such as multiple inheritance. For the most part, you should only use slots on classes that are going to serve as frequently used data structures in your program (e.g., if your program created millions of instances of a particular class).

A common misperception of `__slots__` is that it is an encapsulation tool that prevents users from adding new attributes to instances. Although this is a side effect of using slots, this was never the original purpose. Instead, `__slots__` was always intended to be an optimization tool.

Encapsulating Names in a Class

Problem

You want to encapsulate "private" data on instances of a class, but are concerned about Python's lack of access control.

Solution

Rather than relying on language features to encapsulate data, Python programmers are expected to observe certain naming conventions concerning the intended usage of data and methods. The first convention is that any name that starts with a single leading underscore (_) should always be assumed to be internal implementation. For example:

```
class A:
    def __init__(self):
        self._internal = 0    # An internal attribute
        self.public = 1       # A public attribute

    def public_method(self):
        '''
        A public method
        '''
        ...

    def _internal_method(self):
        ...
```

Python doesn't actually prevent someone from accessing internal names. However, doing so is considered impolite, and may result in fragile code. It should be noted, too, that the use of the leading underscore is also used for module names and module-level functions. For example, if you ever see a module name that starts with a leading underscore (e.g., `_socket`), it's internal implementation. Likewise, module-level functions such as `sys._getframe()` should only be used with great caution.

You may also encounter the use of two leading underscores (__) on names within class definitions. For example:

```
class B:
    def __init__(self):
        self.__private = 0
    def __private_method(self):
        ...
    def public_method(self):
        ...
        self.__private_method()
        ...
```

The use of double leading underscores causes the name to be mangled to something else. Specifically, the private attributes in the preceding class get renamed to `_B__private` and `_B__private_method`, respectively. At this point, you might ask what purpose such name mangling serves. The answer is inheritance—such attributes cannot be overridden via inheritance. For example:

```
class C(B):
    def __init__(self):
        super().__init__()
        self.__private = 1      # Does not override B.__private
    # Does not override B.__private_method()
    def __private_method(self):
        ...
```

Here, the private names `__private` and `__private_method` get renamed to `_C__private` and `_C__private_method`, which are different than the mangled names in the base class `B`.

Discussion

The fact that there are two different conventions (single underscore versus double underscore) for "private" attributes leads to the obvious question of which style you should use. For most code, you should probably just make your nonpublic names start with a single underscore. If, however, you know that your code will involve subclassing, and there are internal attributes that should be hidden from subclasses, use the double underscore instead.

It should also be noted that sometimes you may want to define a variable that clashes with the name of a reserved word. For this, you should use a single trailing underscore. For example:

```
lambda_ = 2.0     # Trailing _ to avoid clash with lambda keyword
```

The reason for not using a leading underscore here is that it avoids confusion about the intended usage (i.e., the use of a leading underscore could be interpreted as a way to avoid a name collision rather than as an indication that the value is private). Using a single trailing underscore solves this problem.

Creating Managed Attributes

Problem

You want to add extra processing (e.g., type checking or validation) to the getting or setting of an instance attribute.

Solution

A simple way to customize access to an attribute is to define it as a "property." For example, this code defines a property that adds simple type checking to an attribute:

```
class Person:
    def __init__(self, first_name):
        self.first_name = first_name

    # Getter function
    @property
    def first_name(self):
        return self._first_name

    # Setter function
    @first_name.setter
    def first_name(self, value):
        if not isinstance(value, str):
            raise TypeError('Expected a string')
        self._first_name = value

    # Deleter function (optional)
    @first_name.deleter
    def first_name(self):
        raise AttributeError("Can't delete attribute")
```

In the preceding code, there are three related methods, all of which must have the same name. The first method is a getter function, and establishes `first_name` as being a property. The other two methods attach optional setter and deleter functions to the `first_name` property. It's important to stress that the `@first_name.setter` and `@first_name.deleter` decorators won't be defined unless `first_name` was already established as a property using `@property`.

A critical feature of a property is that it looks like a normal attribute, but access automatically triggers the getter, setter, and deleter methods. For example:

```
>>> a = Person('Guido')
>>> a.first_name       # Calls the getter
'Guido'
>>> a.first_name = 42  # Calls the setter
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "prop.py", line 14, in first_name
    raise TypeError('Expected a string')
TypeError: Expected a string
>>> del a.first_name
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't delete attribute
>>>
```

When implementing a property, the underlying data (if any) still needs to be stored somewhere. Thus, in the get and set methods, you see direct manipulation of a `_first_name` attribute, which is where the actual data lives. In addition, you may ask why the `__init__()` method sets `self.first_name` instead of `self._first_name`. In this example, the entire point of the property is to apply type checking when setting an attribute. Thus, chances are you would also want such checking to take place during initialization. By setting `self.first_name`, the set operation uses the setter method (as opposed to bypassing it by accessing `self._first_name`).

Properties can also be defined for existing get and set methods. For example:

```
class Person:
    def __init__(self, first_name):
        self.set_first_name(first_name)

    # Getter function
    def get_first_name(self):
        return self._first_name
    # Setter function
    def set_first_name(self, value):
        if not isinstance(value, str):
            raise TypeError('Expected a string')
        self._first_name = value

    # Deleter function (optional)
    def del_first_name(self):
        raise AttributeError("Can't delete attribute")

    # Make a property from existing get/set methods
    name = property(get_first_name, set_first_name, del_first_name)
```

Discussion

A property attribute is actually a collection of methods bundled together. If you inspect a class with a property, you can find the raw methods in the `fget`, `fset`, and `fdel` attributes of the property itself. For example:

```
>>> Person.first_name.fget
<function Person.first_name at 0x1006a60e0>
>>> Person.first_name.fset
<function Person.first_name at 0x1006a6170>
>>> Person.first_name.fdel
<function Person.first_name at 0x1006a62e0>
>>>
```

Normally, you wouldn't call `fget` or `fset` directly, but they are triggered automatically when the property is accessed.

Properties should only be used in cases where you actually need to perform extra processing on attribute access. Sometimes programmers coming from languages such as Java feel that all access should be handled by getters and setters, and that they should write code like this:

```
class Person:
    def __init__(self, first_name):
        self.first_name = name
    @property
    def first_name(self):
        return self._first_name
    @first_name.setter
    def first_name(self, value):
        self._first_name = value
```

Don't write properties that don't actually add anything extra like this. For one, it makes your code more verbose and confusing to others. Second, it will make your program run a lot slower. Lastly, it offers no real design benefit. Specifically, if you later decide that extra processing needs to be added to the handling of an ordinary attribute, you could promote it to a property without changing existing code. This is because the syntax of code that accessed the attribute would remain unchanged.

Properties can also be a way to define computed attributes. These are attributes that are not actually stored, but computed on demand. For example:

```
import math
class Circle:
    def __init__(self, radius):
        self.radius = radius
    @property
    def area(self):
        return math.pi * self.radius ** 2
    @property
    def perimeter(self):
        return 2 * math.pi * self.radius
```

Here, the use of properties results in a very uniform instance interface in that `radius`, `area`, and `perimeter` are all accessed as simple attributes, as opposed to a mix of simple attributes and method calls. For example:

```
>>> c = Circle(4.0)
>>> c.radius
4.0
>>> c.area          # Notice lack of ()
50.26548245743669
>>> c.perimeter     # Notice lack of ()
25.132741228718345
>>>
```

Although properties give you an elegant programming interface, sometimes you actually may want to directly use getter and setter functions. For example:

```
>>> p = Person('Guido')
>>> p.get_first_name()
'Guido'
>>> p.set_first_name('Larry')
>>>
```

This often arises in situations where Python code is being integrated into a larger infrastructure of systems or programs. For example, perhaps a Python class is going to be plugged into a large distributed system based on remote procedure calls or distributed objects. In such a setting, it may be much easier to work with an explicit get/set method (as a normal method call) rather than a property that implicitly makes such calls.

Last, but not least, don’t write Python code that features a lot of repetitive property definitions. For example:

```
class Person:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    @property
    def first_name(self):
        return self._first_name

    @first_name.setter
    def first_name(self, value):
        if not isinstance(value, str):
            raise TypeError('Expected a string')
        self._first_name = value

    # Repeated property code, but for a different name (bad!)
    @property
    def last_name(self):
        return self._last_name

    @last_name.setter
    def last_name(self, value):
        if not isinstance(value, str):
            raise TypeError('Expected a string')
        self._last_name = value
```

Code repetition leads to bloated, error prone, and ugly code. As it turns out, there are much better ways to achieve the same thing using descriptors or closures. See Recipes and .

Calling a Method on a Parent Class

Problem

You want to invoke a method in a parent class in place of a method that has been overridden in a subclass.

Solution

To call a method in a parent (or superclass), use the `super()` function. For example:

```
class A:
    def spam(self):
        print('A.spam')

class B(A):
    def spam(self):
        print('B.spam')
        super().spam()      # Call parent spam()
```

A very common use of `super()` is in the handling of the `__init__()` method to make sure that parents are properly initialized:

```
class A:
    def __init__(self):
        self.x = 0

class B(A):
    def __init__(self):
        super().__init__()
        self.y = 1
```

Another common use of `super()` is in code that overrides any of Python's special methods. For example:

```
class Proxy:
    def __init__(self, obj):
        self._obj = obj

    # Delegate attribute lookup to internal obj
    def __getattr__(self, name):
        return getattr(self._obj, name)

    # Delegate attribute assignment
    def __setattr__(self, name, value):
        if name.startswith('_'):
            super().__setattr__(name, value)    # Call original __setattr__
        else:
            setattr(self._obj, name, value)
```

In this code, the implementation of `__setattr__()` includes a name check. If the name starts with an underscore (_), it invokes the original implementation of `__setattr__()` using `super()`. Otherwise, it delegates to the internally held object `self._obj`. It looks a little funny, but `super()` works even though there is no explicit base class listed.

Discussion

Correct use of the `super()` function is actually one of the most poorly understood aspects of Python. Occasionally, you will see code written that directly calls a method in a parent like this:

```
class Base:
    def __init__(self):
        print('Base.__init__')

class A(Base):
    def __init__(self):
        Base.__init__(self)
        print('A.__init__')
```

Although this "works" for most code, it can lead to bizarre trouble in advanced code involving multiple inheritance. For example, consider the following:

```
class Base:
    def __init__(self):
        print('Base.__init__')

class A(Base):
    def __init__(self):
        Base.__init__(self)
        print('A.__init__')

class B(Base):
    def __init__(self):
        Base.__init__(self)
        print('B.__init__')

class C(A,B):
    def __init__(self):
        A.__init__(self)
        B.__init__(self)
        print('C.__init__')
```

If you run this code, you'll see that the `Base.__init__()` method gets invoked twice, as shown here:

```
>>> c = C()
Base.__init__
A.__init__
Base.__init__
B.__init__
C.__init__
>>>
```

Perhaps double-invocation of `Base.__init__()` is harmless, but perhaps not. If, on the other hand, you change the code to use `super()`, it all works:

```
class Base:
    def __init__(self):
        print('Base.__init__')

class A(Base):
    def __init__(self):
        super().__init__()
        print('A.__init__')

class B(Base):
    def __init__(self):
        super().__init__()
        print('B.__init__')

class C(A,B):
    def __init__(self):
        super().__init__()     # Only one call to super() here
        print('C.__init__')
```

When you use this new version, you'll find that each `__init__()` method only gets called once:

```
>>> c = C()
Base.__init__
B.__init__
A.__init__
C.__init__
>>>
```

To understand why it works, we need to step back for a minute and discuss how Python implements inheritance. For every class that you define, Python computes what's known as a method resolution order (MRO) list. The MRO list is simply a linear ordering of all the base classes. For example:

```
>>> C.__mro__
(<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>,
<class '__main__.Base'>, <class 'object'>)
>>>
```

To implement inheritance, Python starts with the leftmost class and works its way left-to-right through classes on the MRO list until it finds the first attribute match.

The actual determination of the MRO list itself is made using a technique known as C3 Linearization. Without getting too bogged down in the mathematics of it, it is actually a merge sort of the MROs from the parent classes subject to three constraints:

- Child classes get checked before parents
- Multiple parents get checked in the order listed.
- If there are two valid choices for the next class, pick the one from the first parent.

Honestly, all you really need to know is that the order of classes in the MRO list "makes sense" for almost any class hierarchy you are going to define.

When you use the `super()` function, Python continues its search starting with the next class on the MRO. As long as every redefined method consistently uses `super()` and only calls it once, control will ultimately work its way through the entire MRO list and each method will only be called once. This is why you don't get double calls to `Base.__init__()` in the second example.

A somewhat surprising aspect of `super()` is that it doesn't necessarily go to the direct parent of a class next in the MRO and that you can even use it in a class with no direct parent at all. For example, consider this class:

```
class A:
    def spam(self):
        print('A.spam')
        super().spam()
```

If you try to use this class, you'll find that it's completely broken:

```
>>> a = A()
>>> a.spam()
A.spam
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in spam
AttributeError: 'super' object has no attribute 'spam'
>>>
```

Yet, watch what happens if you start using the class with multiple inheritance:

```
>>> class B:
...     def spam(self):
...         print('B.spam')
...
>>> class C(A,B):
...     pass
...
>>> c = C()
>>> c.spam()
A.spam
B.spam
>>>
```

Here you see that the use of `super().spam()` in class A has, in fact, called the `spam()` method in class B —a class that is completely unrelated to A! This is all explained by the MRO of class C:

```
>>> C.__mro__
(<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>,
<class 'object'>)
>>>
```

Using `super()` in this manner is most common when defining mixin classes. See Recipes and .

However, because `super()` might invoke a method that you're not expecting, there are a few general rules of thumb you should try to follow. First, make sure that all methods with the same name in an inheritance hierarchy have a compatible calling signature (i.e., same number of arguments, argument names). This ensures that `super()` won't get tripped up if it tries to invoke a method on a class that's not a direct parent. Second, it's usually a good idea to make sure that the topmost class provides an implementation of the method so that the chain of lookups that occur along the MRO get terminated by an actual method of some sort.

Use of `super()` is sometimes a source of debate in the Python community. However, all things being equal, you should probably use it in modern code. Raymond Hettinger has written an excellent blog post "Python's super() Considered Super!" that has even more examples and reasons why `super()` might be super-awesome.

Extending a Property in a Subclass

Problem

Within a subclass, you want to extend the functionality of a property defined in a parent class.

Solution

Consider the following code, which defines a property:

```
class Person:
    def __init__(self, name):
        self.name = name

    # Getter function
    @property
    def name(self):
        return self._name

    # Setter function
    @name.setter
    def name(self, value):
        if not isinstance(value, str):
            raise TypeError('Expected a string')
        self._name = value

    # Deleter function
    @name.deleter
    def name(self):
        raise AttributeError("Can't delete attribute")
```

Here is an example of a class that inherits from `Person` and extends the `name` property with new functionality:

```
class SubPerson(Person):
    @property
    def name(self):
        print('Getting name')
        return super().name

    @name.setter
    def name(self, value):
        print('Setting name to', value)
        super(SubPerson, SubPerson).name.__set__(self, value)

    @name.deleter
    def name(self):
        print('Deleting name')
        super(SubPerson, SubPerson).name.__delete__(self)
```

Here is an example of the new class in use:

```
>>> s = SubPerson('Guido')
Setting name to Guido
```

```
>>> s.name
Getting name
'Guido'
>>> s.name = 'Larry'
Setting name to Larry
>>> s.name = 42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "example.py", line 16, in name
       raise TypeError('Expected a string')
TypeError: Expected a string
>>>
```

If you only want to extend one of the methods of a property, use code such as the following:

```
class SubPerson(Person):
    @Person.name.getter
    def name(self):
        print('Getting name')
        return super().name
```

Or, alternatively, for just the setter, use this code:

```
class SubPerson(Person):
    @Person.name.setter
    def name(self, value):
        print('Setting name to', value)
        super(SubPerson, SubPerson).name.__set__(self, value)
```

Discussion

Extending a property in a subclass introduces a number of very subtle problems related to the fact that a property is defined as a collection of getter, setter, and deleter methods, as opposed to just a single method. Thus, when extending a property, you need to figure out if you will redefine all of the methods together or just one of the methods.

In the first example, all of the property methods are redefined together. Within each method, `super()` is used to call the previous implementation. The use of `super(SubPerson, SubPerson).name.__set__(self, value)` in the setter function is no mistake. To delegate to the previous implementation of the setter, control needs to pass through the `__set__()` method of the previously defined `name` property. However, the only way to get to this method is to access it as a class variable instead of an instance variable. This is what happens with the `super(SubPerson, SubPerson)` operation.

If you only want to redefine one of the methods, it's not enough to use `@property` by itself. For example, code like this doesn't work:

```
class SubPerson(Person):
    @property              # Doesn't work
    def name(self):
        print('Getting name')
        return super().name
```

If you try the resulting code, you'll find that the setter function disappears entirely:

```
>>> s = SubPerson('Guido')
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "example.py", line 5, in __init__
    self.name = name
AttributeError: can't set attribute
>>>
```

Instead, you should change the code to that shown in the solution:

```
class SubPerson(Person):
    @Person.getter
    def name(self):
        print('Getting name')
        return super().name
```

When you do this, all of the previously defined methods of the property are copied, and the getter function is replaced. It now works as expected:

```
>>> s = SubPerson('Guido')
>>> s.name
Getting name
'Guido'
>>> s.name = 'Larry'
>>> s.name
Getting name
'Larry'
>>> s.name = 42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "example.py", line 16, in name
    raise TypeError('Expected a string')
TypeError: Expected a string
>>>
```

In this particular solution, there is no way to replace the hardcoded class name `Person` with something more generic. If you don't know which base class defined a property, you should use the solution where all of the property methods are redefined and `super()` is used to pass control to the previous implementation.

It's worth noting that the first technique shown in this recipe can also be used to extend a descriptor, as described in "Creating a New Kind of Class or Instance Attribute". For example:

```
# A descriptor
class String:
    def __init__(self, name):
        self.name = name

    def __get__(self, instance, cls):
        if instance is None:
            return self
        return instance.__dict__[self.name]

    def __set__(self, instance, value):
        if not isinstance(value, str):
            raise TypeError('Expected a string')
        instance.__dict__[self.name] = value

# A class with a descriptor
```

```
class Person:
    name = String('name')
    def __init__(self, name):
        self.name = name

# Extending a descriptor with a property
class SubPerson(Person):
    @property
    def name(self):
        print('Getting name')
        return super().name

    @name.setter
    def name(self, value):
        print('Setting name to', value)
        super(SubPerson, SubPerson).name.__set__(self, value)

    @name.deleter
    def name(self):
        print('Deleting name')
        super(SubPerson, SubPerson).name.__delete__(self)
```

Finally, it's worth noting that by the time you read this, subclassing of setter and deleter methods might be somewhat simplified. The solution shown will still work, but the bug reported at Python's issues page might resolve into a cleaner approach in a future Python version.

Creating a New Kind of Class or Instance Attribute

Problem

You want to create a new kind of instance attribute type with some extra functionality, such as type checking.

Solution

If you want to create an entirely new kind of instance attribute, define its functionality in the form of a descriptor class. Here is an example:

```
# Descriptor attribute for an integer type-checked attribute
class Integer:
    def __init__(self, name):
        self.name = name

    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            return instance.__dict__[self.name]

    def __set__(self, instance, value):
        if not isinstance(value, int):
            raise TypeError('Expected an int')
        instance.__dict__[self.name] = value

    def __delete__(self, instance):
        del instance.__dict__[self.name]
```

A descriptor is a class that implements the three core attribute access operations (get, set, and delete) in the form of `__get__()`, `__set__()`, and `__delete__()` special methods. These methods work by receiving an instance as input. The underlying dictionary of the instance is then manipulated as appropriate.

To use a descriptor, instances of the descriptor are placed into a class definition as class variables. For example:

```
class Point:
    x = Integer('x')
    y = Integer('y')
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

When you do this, all access to the descriptor attributes (e.g., `x` or `y`) is captured by the `__get__()`, `__set__()`, and `__delete__()` methods. For example:

```
>>> p = Point(2, 3)
>>> p.x           # Calls Point.x.__get__(p,Point)
2
>>> p.y = 5       # Calls Point.y.__set__(p, 5)
>>> p.x = 2.3     # Calls Point.x.__set__(p, 2.3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "descrip.py", line 12, in __set__
    raise TypeError('Expected an int')
TypeError: Expected an int
>>>
```

As input, each method of a descriptor receives the instance being manipulated. To carry out the requested operation, the underlying instance dictionary (the `__dict__` attribute) is manipulated as appropriate. The `self.name` attribute of the descriptor holds the dictionary key being used to store the actual data in the instance dictionary.

Discussion

Descriptors provide the underlying magic for most of Python's class features, including `@classmethod`, `@staticmethod`, `@property`, and even the `__slots__` specification.

By defining a descriptor, you can capture the core instance operations (get, set, delete) at a very low level and completely customize what they do. This gives you great power, and is one of the most important tools employed by the writers of advanced libraries and frameworks.

One confusion with descriptors is that they can only be defined at the class level, not on a per-instance basis. Thus, code like this will not work:

```
# Does NOT work
class Point:
    def __init__(self, x, y):
        self.x = Integer('x')    # No! Must be a class variable
        self.y = Integer('y')
        self.x = x
        self.y = y
```

Also, the implementation of the `__get__()` method is trickier than it seems:

```
# Descriptor attribute for an integer type-checked attribute
class Integer:
    ...
    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            return instance.__dict__[self.name]
    ...
```

The reason `__get__()` looks somewhat complicated is to account for the distinction between instance variables and class variables. If a descriptor is accessed as a class variable, the `instance` argument is set to `None`. In this case, it is standard practice to simply return the descriptor instance itself (although any kind of custom processing is also allowed). For example:

```
>>> p = Point(2,3)
>>> p.x      # Calls Point.x.__get__(p, Point)
2
>>> Point.x  # Calls Point.x.__get__(None, Point)
<__main__.Integer object at 0x100671890>
>>>
```

Descriptors are often just one component of a larger programming framework involving decorators or metaclasses. As such, their use may be hidden just barely out of sight. As an example, here is some more advanced descriptor-based code involving a class decorator:

```
# Descriptor for a type-checked attribute
class Typed:
    def __init__(self, name, expected_type):
        self.name = name
        self.expected_type = expected_type

    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            return instance.__dict__[self.name]

    def __set__(self, instance, value):
        if not isinstance(value, self.expected_type):
            raise TypeError('Expected ' + str(self.expected_type))
        instance.__dict__[self.name] = value

    def __delete__(self, instance):
        del instance.__dict__[self.name]

# Class decorator that applies it to selected attributes
def typeassert(**kwargs):
    def decorate(cls):
        for name, expected_type in kwargs.items():
            # Attach a Typed descriptor to the class
            setattr(cls, name, Typed(name, expected_type))
        return cls
    return decorate

# Example use
@typeassert(name=str, shares=int, price=float)
```

```
class Stock:
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price
```

Finally, it should be stressed that you would probably not write a descriptor if you simply want to customize the access of a single attribute of a specific class. For that, it's easier to use a property instead, as described in "Creating Managed Attributes". Descriptors are more useful in situations where there will be a lot of code reuse (i.e., you want to use the functionality provided by the descriptor in hundreds of places in your code or provide it as a library feature).

Using Lazily Computed Properties

Problem

You'd like to define a read-only attribute as a property that only gets computed on access. However, once accessed, you'd like the value to be cached and not recomputed on each access.

Solution

An efficient way to define a lazy attribute is through the use of a descriptor class, such as the following:

```
class lazyproperty:
    def __init__(self, func):
        self.func = func

    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            value = self.func(instance)
            setattr(instance, self.func.__name__, value)
            return value
```

To utilize this code, you would use it in a class such as the following:

```
import math

class Circle:
    def __init__(self, radius):
        self.radius = radius

    @lazyproperty
    def area(self):
        print('Computing area')
        return math.pi * self.radius ** 2

    @lazyproperty
    def perimeter(self):
        print('Computing perimeter')
        return 2 * math.pi * self.radius
```

Here is an interactive session that illustrates how it works:

```
>>> c = Circle(4.0)
>>> c.radius
4.0
>>> c.area
Computing area
50.26548245743669
>>> c.area
50.26548245743669
>>> c.perimeter
Computing perimeter
25.132741228718345
>>> c.perimeter
25.132741228718345
>>>
```

Carefully observe that the messages "Computing area" and "Computing perimeter" only appear once.

Discussion

In many cases, the whole point of having a lazily computed attribute is to improve performance. For example, you avoid computing values unless you actually need them somewhere. The solution shown does just this, but it exploits a subtle feature of descriptors to do it in a highly efficient way.

As shown in other recipes (e.g., "Creating a New Kind of Class or Instance Attribute"), when a descriptor is placed into a class definition, its `__get__()`, `__set__()`, and `__delete__()` methods get triggered on attribute access. However, if a descriptor only defines a `__get__()` method, it has a much weaker binding than usual. In particular, the `__get__()` method only fires if the attribute being accessed is not in the underlying instance dictionary.

The `lazyproperty` class exploits this by having the `__get__()` method store the computed value on the instance using the same name as the property itself. By doing this, the value gets stored in the instance dictionary and disables further computation of the property. You can observe this by digging a little deeper into the example:

```
>>> c = Circle(4.0)
>>> # Get instance variables
>>> vars(c)
{'radius': 4.0}

>>> # Compute area and observe variables afterward
>>> c.area
Computing area
50.26548245743669
>>> vars(c)
{'area': 50.26548245743669, 'radius': 4.0}

>>> # Notice access doesn't invoke property anymore
>>> c.area
50.26548245743669

>>> # Delete the variable and see property trigger again
>>> del c.area
>>> vars(c)
{'radius': 4.0}
>>> c.area
Computing area
50.26548245743669
```

```
>>>
```

One possible downside to this recipe is that the computed value becomes mutable after it's created. For example:

```
>>> c.area
Computing area
50.26548245743669
>>> c.area = 25
>>> c.area
25
>>>
```

If that's a concern, you can use a slightly less efficient implementation, like this:

```
def lazyproperty(func):
    name = '_lazy_' + func.__name__
    @property
    def lazy(self):
        if hasattr(self, name):
            return getattr(self, name)
        else:
            value = func(self)
            setattr(self, name, value)
            return value
    return lazy
```

If you use this version, you'll find that set operations are not allowed. For example:

```
>>> c = Circle(4.0)
>>> c.area
Computing area
50.26548245743669
>>> c.area
50.26548245743669
>>> c.area = 25
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>>
```

However, a disadvantage is that all get operations have to be routed through the property's getter function. This is less efficient than simply looking up the value in the instance dictionary, as was done in the original solution.

For more information on properties and managed attributes, see "Creating Managed Attributes". Descriptors are described in "Creating a New Kind of Class or Instance Attribute".

Simplifying the Initialization of Data Structures

Problem

You are writing a lot of classes that serve as data structures, but you are getting tired of writing highly repetitive and boilerplate `__init__()` functions.

Solution

You can often generalize the initialization of data structures into a single `__init__()` function defined in a common base class. For example:

```
class Structure:
    # Class variable that specifies expected fields
    _fields= []
    def __init__(self, *args):
        if len(args) != len(self._fields):
            raise TypeError('Expected {} arguments'.format(len(self._fields)))

        # Set the arguments
        for name, value in zip(self._fields, args):
            setattr(self, name, value)


# Example class definitions
if __name__ == '__main__':
    class Stock(Structure):
        _fields = ['name', 'shares', 'price']

    class Point(Structure):
        _fields = ['x','y']

    class Circle(Structure):
        _fields = ['radius']
        def area(self):
            return math.pi * self.radius ** 2
```

If you use the resulting classes, you'll find that they are easy to construct. For example:

```
>>> s = Stock('ACME', 50, 91.1)
>>> p = Point(2, 3)
>>> c = Circle(4.5)
>>> s2 = Stock('ACME', 50)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "structure.py", line 6, in __init__
    raise TypeError('Expected {} arguments'.format(len(self._fields)))
TypeError: Expected 3 arguments
```

Should you decide to support keyword arguments, there are several design options. One choice is to map the keyword arguments so that they only correspond to the attribute names specified in `_fields`. For example:

```
class Structure:
    _fields= []
    def __init__(self, *args, **kwargs):
        if len(args) > len(self._fields):
            raise TypeError('Expected {} arguments'.format(len(self._fields)))

        # Set all of the positional arguments
        for name, value in zip(self._fields, args):
            setattr(self, name, value)

        # Set the remaining keyword arguments
        for name in self._fields[len(args):]:
                setattr(self, name, kwargs.pop(name))

        # Check for any remaining unknown arguments
        if kwargs:
            raise TypeError('Invalid argument(s): {}'.format(','.join(kwargs)))

# Example use
if __name__ == '__main__':
    class Stock(Structure):
        _fields = ['name', 'shares', 'price']

    s1 = Stock('ACME', 50, 91.1)
    s2 = Stock('ACME', 50, price=91.1)
    s3 = Stock('ACME', shares=50, price=91.1)
```

Another possible choice is to use keyword arguments as a means for adding additional attributes to the structure not specified in `_fields`. For example:

```
class Structure:
    # Class variable that specifies expected fields
    _fields= []
    def __init__(self, *args, **kwargs):
        if len(args) != len(self._fields):
            raise TypeError('Expected {} arguments'.format(len(self._fields)))

        # Set the arguments
        for name, value in zip(self._fields, args):
            setattr(self, name, value)

        # Set the additional arguments (if any)
        extra_args = kwargs.keys() - self._fields
        for name in extra_args:
            setattr(self, name, kwargs.pop(name))
        if kwargs:
            raise TypeError('Duplicate values for {}'.format(','.join(kwargs)))

# Example use
if __name__ == '__main__':
    class Stock(Structure):
        _fields = ['name', 'shares', 'price']

    s1 = Stock('ACME', 50, 91.1)
    s2 = Stock('ACME', 50, 91.1, date='8/2/2012')
```

Discussion

This technique of defining a general purpose `__init__()` method can be extremely useful if you're ever writing a program built around a large number of small data structures. It leads to much less code than manually writing `__init__()` methods like this:

```
class Stock:
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

class Circle:
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        return math.pi * self.radius ** 2
```

One subtle aspect of the implementation concerns the mechanism used to set value using the `setattr()` function. Instead of doing that, you might be inclined to directly access the instance dictionary. For example:

```
class Structure:
    # Class variable that specifies expected fields
    _fields= []
    def __init__(self, *args):
        if len(args) != len(self._fields):
            raise TypeError('Expected {} arguments'.format(len(self._fields)))

        # Set the arguments (alternate)
        self.__dict__.update(zip(self._fields,args))
```

Although this works, it's often not safe to make assumptions about the implementation of a subclass. If a subclass decided to use `__slots__` or wrap a specific attribute with a property (or descriptor), directly acccessing the instance dictionary would break. The solution has been written to be as general purpose as possible and not to make any assumptions about subclasses.

A potential downside of this technique is that it impacts documentation and help features of IDEs. If a user asks for help on a specific class, the required arguments aren't described in the usual way. For example:

```
>>> help(Stock)
Help on class Stock in module __main__:

class Stock(Structure)
...
 |  Methods inherited from Structure:
 |
 |  __init__(self, *args, **kwargs)
 |
...
>>>
```

Many of these problems can be fixed by either attaching or enforcing a type signature in the `__init__()` function. See "Enforcing an Argument Signature on *args and **kwargs".

It should be noted that it is also possible to automatically initialize instance variables using a utility function and a so-called "frame hack." For example:

```
def init_fromlocals(self):
    import sys
    locs = sys._getframe(1).f_locals
    for k, v in locs.items():
        if k != 'self':
            setattr(self, k, v)
```

```
class Stock:
    def __init__(self, name, shares, price):
        init_fromlocals(self)
```

In this variation, the `init_fromlocals()` function uses `sys._getframe()` to peek at the local variables of the calling method. If used as the first step of an `__init__()` method, the local variables will be the same as the passed arguments and can be easily used to set attributes with the same names. Although this approach avoids the problem of getting the right calling signature in IDEs, it runs more than 50% slower than the solution provided in the recipe, requires more typing, and involves more sophisticated magic behind the scenes. If your code doesn't need this extra power, often times the simpler solution will work just fine.

Defining an Interface or Abstract Base Class

Problem

You want to define a class that serves as an interface or abstract base class from which you can perform type checking and ensure that certain methods are implemented in subclasses.

Solution

To define an abstract base class, use the `abc` module. For example:

```
from abc import ABCMeta, abstractmethod

class IStream(metaclass=ABCMeta):
    @abstractmethod
    def read(self, maxbytes=-1):
        pass
    @abstractmethod
    def write(self, data):
        pass
```

A central feature of an abstract base class is that it cannot be instantiated directly. For example, if you try to do it, you'll get an error:

```
a = IStream()   # TypeError: Can't instantiate abstract class
                # IStream with abstract methods read, write
```

Instead, an abstract base class is meant to be used as a base class for other classes that are expected to implement the required methods. For example:

```
class SocketStream(IStream):
    def read(self, maxbytes=-1):
        ...
    def write(self, data):
        ...
```

A major use of abstract base classes is in code that wants to enforce an expected programming interface. For example, one way to view the `IStream` base class is as a high-level specification for an interface that allows reading and writing of data. Code that explicitly checks for this interface could be written as follows:

```
def serialize(obj, stream):
    if not isinstance(stream, IStream):
        raise TypeError('Expected an IStream')
    ...
```

You might think that this kind of type checking only works by subclassing the abstract base class (ABC), but ABCs allow other classes to be registered as implementing the required interface. For example, you can do this:

```
import io

# Register the built-in I/O classes as supporting our interface
IStream.register(io.IOBase)

# Open a normal file and type check
f = open('foo.txt')
isinstance(f, IStream)      # Returns True
```

It should be noted that `@abstractmethod` can also be applied to static methods, class methods, and properties. You just need to make sure you apply it in the proper sequence where `@abstractmethod` appears immediately before the function definition, as shown here:

```
from abc import ABCMeta, abstractmethod

class A(metaclass=ABCMeta):
    @property
    @abstractmethod
    def name(self):
        pass

    @name.setter
    @abstractmethod
    def name(self, value):
        pass

    @classmethod
    @abstractmethod
    def method1(cls):
        pass

    @staticmethod
    @abstractmethod
    def method2():
        pass
```

Discussion

Predefined abstract base classes are found in various places in the standard library. The `collections` module defines a variety of ABCs related to containers and iterators (sequences, mappings, sets, etc.), the `numbers` library defines ABCs related to numeric objects (integers, floats, rationals, etc.), and the `io` library defines ABCs related to I/O handling.

You can use the predefined ABCs to perform more generalized kinds of type checking. Here are some examples:

```
import collections
```

```
# Check if x is a sequence
if isinstance(x, collections.Sequence):
    ...

# Check if x is iterable
if isinstance(x, collections.Iterable):
    ...

# Check if x has a size
if isinstance(x, collections.Sized):
    ...

# Check if x is a mapping
if isinstance(x, collections.Mapping):
    ...
```

It should be noted that, as of this writing, certain library modules don't make use of these predefined ABCs as you might expect. For example:

```
from decimal import Decimal
import numbers

x = Decimal('3.4')
isinstance(x, numbers.Real)   # Returns False
```

Even though the value `3.4` is technically a real number, it doesn't type check that way to help avoid inadvertent mixing of floating-point numbers and decimals. Thus, if you use the ABC functionality, it is wise to carefully write tests that verify that the behavior is as you intended.

Although ABCs facilitate type checking, it's not something that you should overuse in a program. At its heart, Python is a dynamic language that gives you great flexibility. Trying to enforce type constraints everywhere tends to result in code that is more complicated than it needs to be. You should embrace Python's flexibility.

Implementing a Data Model or Type System

Problem

You want to define various kinds of data structures, but want to enforce constraints on the values that are allowed to be assigned to certain attributes.

Solution

In this problem, you are basically faced with the task of placing checks or assertions on the setting of certain instance attributes. To do this, you need to customize the setting of attributes on a per-attribute basis. To do this, you should use descriptors.

The following code illustrates the use of descriptors to implement a system type and value checking framework:

```
# Base class. Uses a descriptor to set a value
class Descriptor:
    def __init__(self, name=None, **opts):
        self.name = name
        for key, value in opts.items():
            setattr(self, key, value)

    def __set__(self, instance, value):
        instance.__dict__[self.name] = value

# Descriptor for enforcing types
class Typed(Descriptor):
    expected_type = type(None)

    def __set__(self, instance, value):
        if not isinstance(value, self.expected_type):
            raise TypeError('expected ' + str(self.expected_type))
        super().__set__(instance, value)

# Descriptor for enforcing values
class Unsigned(Descriptor):
    def __set__(self, instance, value):
        if value < 0:
            raise ValueError('Expected >= 0')
        super().__set__(instance, value)

class MaxSized(Descriptor):
    def __init__(self, name=None, **opts):
        if 'size' not in opts:
            raise TypeError('missing size option')
        super().__init__(name, **opts)

    def __set__(self, instance, value):
        if len(value) >= self.size:
            raise ValueError('size must be < ' + str(self.size))
        super().__set__(instance, value)
```

These classes should be viewed as basic building blocks from which you construct a data model or type system. Continuing, here is some code that implements some different kinds of data:

```
class Integer(Typed):
    expected_type = int

class UnsignedInteger(Integer, Unsigned):
    pass

class Float(Typed):
    expected_type = float

class UnsignedFloat(Float, Unsigned):
    pass

class String(Typed):
    expected_type = str

class SizedString(String, MaxSized):
    pass
```

Using these type objects, it is now possible to define a class such as this:

```
class Stock:
    # Specify constraints
    name = SizedString('name',size=8)
    shares = UnsignedInteger('shares')
    price = UnsignedFloat('price')
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price
```

With the constraints in place, you'll find that assigning of attributes is now validated. For example:

```
>>> s = Stock('ACME', 50, 91.1)
>>> s.name
'ACME'
>>> s.shares = 75
>>> s.shares = -10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "example.py", line 17, in __set__
    super().__set__(instance, value)
  File "example.py", line 23, in __set__
    raise ValueError('Expected >= 0')
ValueError: Expected >= 0
>>> s.price = 'a lot'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "example.py", line 16, in __set__
    raise TypeError('expected ' + str(self.expected_type))
TypeError: expected <class 'float'>
>>> s.name = 'ABRACADABRA'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "example.py", line 17, in __set__
    super().__set__(instance, value)
  File "example.py", line 35, in __set__
    raise ValueError('size must be < ' + str(self.size))
ValueError: size must be < 8
>>>
```

There are some techniques that can be used to simplify the specification of constraints in classes. One approach is to use a class decorator, like this:

```
# Class decorator to apply constraints
def check_attributes(**kwargs):
    def decorate(cls):
        for key, value in kwargs.items():
            if isinstance(value, Descriptor):
                value.name = key
                setattr(cls, key, value)
            else:
                setattr(cls, key, value(key))
        return cls
    return decorate

# Example
@check_attributes(name=SizedString(size=8),
                  shares=UnsignedInteger,
                  price=UnsignedFloat)
class Stock:
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price
```

Another approach to simplify the specification of constraints is to use a metaclass. For example:

```
# A metaclass that applies checking
class checkedmeta(type):
    def __new__(cls, clsname, bases, methods):
        # Attach attribute names to the descriptors
        for key, value in methods.items():
            if isinstance(value, Descriptor):
                value.name = key
        return type.__new__(cls, clsname, bases, methods)


# Example
class Stock(metaclass=checkedmeta):
    name   = SizedString(size=8)
    shares = UnsignedInteger()
    price  = UnsignedFloat()
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price
```

Discussion

This recipe involves a number of advanced techniques, including descriptors, mixin classes, the use of `super()`, class decorators, and metaclasses. Covering the basics of all those topics is beyond what can be covered here, but examples can be found in other recipes (see Recipes , , , and). However, there are a number of subtle points worth noting.

First, in the `Descriptor` base class, you will notice that there is a `__set__()` method, but no corresponding `__get__()`. If a descriptor will do nothing more than extract an identically named value from the underlying instance dictionary, defining `__get__()` is unnecessary. In fact, defining `__get__()` will just make it run slower. Thus, this recipe only focuses on the implementation of `__set__()`.

The overall design of the various descriptor classes is based on mixin classes. For example, the `Unsigned` and `MaxSized` classes are meant to be mixed with the other descriptor classes derived from `Typed`. To handle a specific kind of data type, multiple inheritance is used to combine the desired functionality.

You will also notice that all `__init__()` methods of the various descriptors have been programmed to have an identical signature involving keyword arguments `**opts`. The class for `MaxSized` looks for its required attribute in `opts`, but simply passes it along to the `Descriptor` base class, which actually sets it. One tricky part about composing classes like this (especially mixins), is that you don't always know how the classes are going to be chained together or what `super()` will invoke. For this reason, you need to make it work with any possible combination of classes.

The definitions of the various type classes such as `Integer`, `Float`, and `String` illustrate a useful technique of using class variables to customize an implementation. The `Typed` descriptor merely looks for an `expected_type` attribute that is provided by each of those subclasses.

The use of a class decorator or metaclass is often useful for simplifying the specification by the user. You will notice that in those examples, the user no longer has to type the name of the attribute more than once. For example:

```
# Normal
```

```
class Point:
    x = Integer('x')
    y = Integer('y')

# Metaclass
class Point(metaclass=checkedmeta):
    x = Integer()
    y = Integer()
```

The code for the class decorator and metaclass simply scan the class dictionary looking for descriptors. When found, they simply fill in the descriptor name based on the key value.

Of all the approaches, the class decorator solution may provide the most flexibility and sanity. For one, it does not rely on any advanced machinery, such as metaclasses. Second, decoration is something that can easily be added or removed from a class definition as desired. For example, within the decorator, there could be an option to simply omit the added checking altogether. These might allow the checking to be something that could be turned on or off depending on demand (maybe for debugging versus production).

As a final twist, a class decorator approach can also be used as a replacement for mixin classes, multiple inheritance, and tricky use of the `super()` function. Here is an alternative formulation of this recipe that uses class decorators:

```
# Base class. Uses a descriptor to set a value
class Descriptor:
    def __init__(self, name=None, **opts):
        self.name = name
        for key, value in opts.items():
            setattr(self, key, value)

    def __set__(self, instance, value):
        instance.__dict__[self.name] = value

# Decorator for applying type checking
def Typed(expected_type, cls=None):
    if cls is None:
        return lambda cls: Typed(expected_type, cls)

    super_set = cls.__set__
    def __set__(self, instance, value):
        if not isinstance(value, expected_type):
            raise TypeError('expected ' + str(expected_type))
        super_set(self, instance, value)
    cls.__set__ = __set__
    return cls

# Decorator for unsigned values
def Unsigned(cls):
    super_set = cls.__set__
    def __set__(self, instance, value):
        if value < 0:
            raise ValueError('Expected >= 0')
        super_set(self, instance, value)
    cls.__set__ = __set__
    return cls

# Decorator for allowing sized values
def MaxSized(cls):
    super_init = cls.__init__
    def __init__(self, name=None, **opts):
        if 'size' not in opts:
            raise TypeError('missing size option')
        super_init(self, name, **opts)
    cls.__init__ = __init__

    super_set = cls.__set__
    def __set__(self, instance, value):
        if len(value) >= self.size:
            raise ValueError('size must be < ' + str(self.size))
        super_set(self, instance, value)
    cls.__set__ = __set__
    return cls

# Specialized descriptors
@Typed(int)
class Integer(Descriptor):
    pass

@Unsigned
class UnsignedInteger(Integer):
    pass

@Typed(float)
class Float(Descriptor):
    pass

@Unsigned
class UnsignedFloat(Float):
    pass

@Typed(str)
class String(Descriptor):
    pass

@MaxSized
class SizedString(String):
    pass
```

The classes defined in this alternative formulation work in exactly the same manner as before (none of the earlier example code changes) except that everything runs much faster. For example, a simple timing test of setting a typed attribute reveals that the class decorator approach runs almost 100% faster than the approach using mixins. Now aren't you glad you read all the way to the end?

Implementing Custom Containers

Problem

You want to implement a custom class that mimics the behavior of a common built-in container type, such as a list or dictionary. However, you're not entirely sure what methods need to be implemented to do it.

Solution

The `collections` library defines a variety of abstract base classes that are extremely useful when implementing custom container classes. To illustrate, suppose you want your class to support iteration. To do that, simply start by having it inherit from `collections.Iterable`, as follows:

```
import collections

class A(collections.Iterable):
    pass
```

The special feature about inheriting from `collections.Iterable` is that it ensures you implement all of the required special methods. If you don't, you'll get an error upon instantiation:

```
>>> a = A()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class A with abstract methods __iter__
>>>
```

To fix this error, simply give the class the required `__iter__()` method and implement it as desired (see Recipes and).

Other notable classes defined in `collections` include `Sequence`, `MutableSequence`, `Mapping`, `MutableMapping`, `Set`, and `MutableSet`. Many of these classes form hierarchies with increasing levels of functionality (e.g., one such hierarchy is `Container`, `Iterable`, `Sized`, `Sequence`, and `MutableSequence`). Again, simply instantiate any of these classes to see what methods need to be implemented to make a custom container with that behavior:

```
>>> import collections
>>> collections.Sequence()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Sequence with abstract methods \
__getitem__, __len__
>>>
```

Here is a simple example of a class that implements the preceding methods to create a sequence where items are always stored in sorted order (it's not a particularly efficient implementation, but it illustrates the general idea):

```
import collections
import bisect

class SortedItems(collections.Sequence):
    def __init__(self, initial=None):
        self._items = sorted(initial) if initial is None else []

    # Required sequence methods
    def __getitem__(self, index):
        return self._items[index]

    def __len__(self):
        return len(self._items)

    # Method for adding an item in the right location
    def add(self, item):
        bisect.insort(self._items, item)
```

Here's an example of using this class:

```
>>> items = SortedItems([5, 1, 3])
>>> list(items)
```

```
[1, 3, 5]
>>> items[0]
1
>>> items[-1]
5
>>> items.add(2)
>>> list(items)
[1, 2, 3, 5]
>>> items.add(-10)
>>> list(items)
[-10, 1, 2, 3, 5]
>>> items[1:4]
[1, 2, 3]
>>> 3 in items
True
>>> len(items)
5
>>> for n in items:
...     print(n)
...
-10
1
2
3
5
>>>
```

As you can see, instances of `SortedItems` behave exactly like a normal sequence and support all of the usual operations, including indexing, iteration, `len()`, containment (the `in` operator), and even slicing.

As an aside, the `bisect` module used in this recipe is a convenient way to keep items in a list sorted. The `bisect.insort()` inserts an item into a list so that the list remains in order.

Discussion

Inheriting from one of the abstract base classes in `collections` ensures that your custom container implements all of the required methods expected of the container. However, this inheritance also facilitates type checking.

For example, your custom container will satisfy various type checks like this:

```
>>> items = SortedItems()
>>> import collections
>>> isinstance(items, collections.Iterable)
True
>>> isinstance(items, collections.Sequence)
True
>>> isinstance(items, collections.Container)
True
>>> isinstance(items, collections.Sized)
True
>>> isinstance(items, collections.Mapping)
False
>>>
```

Many of the abstract base classes in `collections` also provide default implementations of common container methods. To illustrate, suppose you have a class that inherits from `collections.MutableSequence`, like this:

```
class Items(collections.MutableSequence):
    def __init__(self, initial=None):
        self._items = list(initial) if initial is None else []

    # Required sequence methods
    def __getitem__(self, index):
        print('Getting:', index)
        return self._items[index]

    def __setitem__(self, index, value):
        print('Setting:', index, value)
        self._items[index] = value

    def __delitem__(self, index):
        print('Deleting:', index)
        del self._items[index]

    def insert(self, index, value):
        print('Inserting:', index, value)
        self._items.insert(index, value)

    def __len__(self):
        print('Len')
        return len(self._items)
```

If you create an instance of `Items`, you'll find that it supports almost all of the core list methods (e.g., `append()`, `remove()`, `count()`, etc.). These methods are implemented in such a way that they only use the required ones. Here's an interactive session that illustrates this:

```
>>> a = Items([1, 2, 3])
>>> len(a)
Len
3
>>> a.append(4)
Len
Inserting: 3 4
>>> a.append(2)
Len
Inserting: 4 2
>>> a.count(2)
Getting: 0
Getting: 1
Getting: 2
Getting: 3
Getting: 4
Getting: 5
2
>>> a.remove(3)
Getting: 0
Getting: 1
Getting: 2
Deleting: 2
>>>
```

This recipe only provides a brief glimpse into Python's abstract class functionality. The `numbers` module provides a similar collection of abstract classes related to numeric data types. See "Defining an Interface or Abstract Base Class" for more information about making your own abstract base classes.

Delegating Attribute Access

Problem

You want an instance to delegate attribute access to an internally held instance possibly as an alternative to inheritance or in order to implement a proxy.

Solution

Simply stated, delegation is a programming pattern where the responsibility for implementing a particular operation is handed off (i.e., delegated) to a different object. In its simplest form, it often looks something like this:

```
class A:
    def spam(self, x):
        pass

    def foo(self):
        pass

class B:
    def __init__(self):
        self._a = A()

    def spam(self, x):
        # Delegate to the internal self._a instance
        return self._a.spam(x)

    def foo(self):
        # Delegate to the internal self._a instance
        return self._a.foo()

    def bar(self):
        pass
```

If there are only a couple of methods to delegate, writing code such as that just given is easy enough. However, if there are many methods to delegate, an alternative approach is to define the `__getattr__()` method, like this:

```
class A:
    def spam(self, x):
        pass

    def foo(self):
        pass

class B:
    def __init__(self):
        self._a = A()

    def bar(self):
        pass

    # Expose all of the methods defined on class A
    def __getattr__(self, name):
        return getattr(self._a, name)
```

The `__getattr__()` method is kind of like a catch-all for attribute lookup. It's a method that gets called if code tries to access an attribute that doesn't exist. In the preceding code, it would catch access to undefined methods on `B` and simply delegate them to A. For example:

```
b = B()
b.bar()    # Calls B.bar() (exists on B)
b.spam(42) # Calls B.__getattr__('spam') and delegates to A.spam
```

Another example of delegation is in the implementation of proxies. For example:

```
# A proxy class that wraps around another object, but
# exposes its public attributes

class Proxy:
    def __init__(self, obj):
        self._obj = obj

    # Delegate attribute lookup to internal obj
    def __getattr__(self, name):
        print('getattr:', name)
        return getattr(self._obj, name)

    # Delegate attribute assignment
    def __setattr__(self, name, value):
        if name.startswith('_'):
            super().__setattr__(name, value)
        else:
            print('setattr:', name, value)
            setattr(self._obj, name, value)

    # Delegate attribute deletion
    def __delattr__(self, name):
        if name.startswith('_'):
            super().__delattr__(name)
        else:
            print('delattr:', name)
            delattr(self._obj, name)
```

To use this proxy class, you simply wrap it around another instance. For example:

```
class Spam:
    def __init__(self, x):
        self.x = x
    def bar(self, y):
        print('Spam.bar:', self.x, y)

# Create an instance
s = Spam(2)

# Create a proxy around it
p = Proxy(s)

# Access the proxy
print(p.x)     # Outputs 2
p.bar(3)       # Outputs "Spam.bar: 2 3"
p.x = 37       # Changes s.x to 37
```

By customizing the implementation of the attribute access methods, you could customize the proxy to behave in different ways (e.g., logging access, only allowing read-only access, etc.).

Discussion

Delegation is sometimes used as an alternative to inheritance. For example, instead of writing code like this:

```
class A:
    def spam(self, x):
        print('A.spam', x)

    def foo(self):
        print('A.foo')

class B(A):
    def spam(self, x):
        print('B.spam')
        super().spam(x)

    def bar(self):
        print('B.bar')
```

A solution involving delegation would be written as follows:

```
class A:
    def spam(self, x):
        print('A.spam', x)

    def foo(self):
        print('A.foo')

class B:
    def __init__(self):
        self._a = A()

    def spam(self, x):
        print('B.spam', x)
        self._a.spam(x)

    def bar(self):
        print('B.bar')

    def __getattr__(self, name):
        return getattr(self._a, name)
```

This use of delegation is often useful in situations where direct inheritance might not make much sense or where you want to have more control of the relationship between objects (e.g., only exposing certain methods, implementing interfaces, etc.).

When using delegation to implement proxies, there are a few additional details to note. First, the `__getattr__()` method is actually a fallback method that only gets called when an attribute is not found. Thus, when attributes of the proxy instance itself are accessed (e.g., the `_obj` attribute), this method would not be triggered. Second, the `__setattr__()` and `__delattr__()` methods need a bit of extra logic added to separate attributes from the proxy instance inself and attributes on the internal object `_obj`. A common convention is for proxies to only delegate to attributes that don't start with a leading underscore (i.e., proxies only expose the "public" attributes of the held instance).

It is also important to emphasize that the `__getattr__()` method usually does not apply to most special methods that start and end with double underscores. For example, consider this class:

```
class ListLike:
    def __init__(self):
        self._items = []
    def __getattr__(self, name):
        return getattr(self._items, name)
```

If you try to make a `ListLike` object, you'll find that it supports the common list methods, such as `append()` and `insert()`. However, it does not support any of the operators like `len()`, item lookup, and so forth. For example:

```
>>> a = ListLike()
>>> a.append(2)
>>> a.insert(0, 1)
>>> a.sort()
>>> len(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'ListLike' has no len()
>>> a[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'ListLike' object does not support indexing
>>>
```

To support the different operators, you have to manually delegate the associated special methods yourself. For example:

```
class ListLike:
    def __init__(self):
        self._items = []
    def __getattr__(self, name):
        return getattr(self._items, name)

    # Added special methods to support certain list operations
    def __len__(self):
        return len(self._items)
    def __getitem__(self, index):
        return self._items[index]
    def __setitem__(self, index, value):
        self._items[index] = value
    def __delitem__(self, index):
        del self._items[index]
```

See "Implementing Remote Procedure Calls" for another example of using delegation in the context of creating proxy classes for remote procedure call.

Defining More Than One Constructor in a Class

Problem

You're writing a class, but you want users to be able to create instances in more than the one way provided by `__init__()`.

Solution

To define a class with more than one constructor, you should use a class method. Here is a simple example:

```
import time

class Date:
    # Primary constructor
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

    # Alternate constructor
    @classmethod
    def today(cls):
        t = time.localtime()
        return cls(t.tm_year, t.tm_mon, t.tm_mday)
```

To use the alternate constructor, you simply call it as a function, such as `Date.today()`. Here is an example:

```
a = Date(2012, 12, 21)      # Primary
b = Date.today()            # Alternate
```

Discussion

One of the primary uses of class methods is to define alternate constructors, as shown in this recipe. A critical feature of a class method is that it receives the class as the first argument (`cls`). You will notice that this class is used within the method to create and return the final instance. It is extremely subtle, but this aspect of class methods makes them work correctly with features such as inheritance. For example:

```
class NewDate(Date):
    pass

c = Date.today()      # Creates an instance of Date (cls=Date)
d = NewDate.today()   # Creates an instance of NewDate (cls=NewDate)
```

When defining a class with multiple constructors, you should make the `__init__()` function as simple as possible—doing nothing more than assigning attributes from given values. Alternate constructors can then choose to perform advanced operations if needed.

Instead of defining a separate class method, you might be inclined to implement the `__init__()` method in a way that allows for different calling conventions. For example:

```
class Date:
    def __init__(self, *args):
        if len(args) == 0:
            t = time.localtime()
            args = (t.tm_year, t.tm_mon, t.tm_mday)
        self.year, self.month, self.day = args
```

Although this technique works in certain cases, it often leads to code that is hard to understand and difficult to maintain. For example, this implementation won't show useful help strings (with argument names). In addition, code that creates `Date` instances will be less clear. Compare and contrast the following:

```
a = Date(2012, 12, 21)   # Clear. A specific date.
b = Date()               # ??? What does this do?

# Class method version
c = Date.today()         # Clear. Today's date.
```

As shown, the `Date.today()` invokes the regular `Date.__init__()` method by instantiating a `Date()` with suitable year, month, and day arguments. If necessary, instances can be created without ever invoking the `__init__()` method. This is described in the next recipe.

Creating an Instance Without Invoking *init*

Problem

You need to create an instance, but want to bypass the execution of the `__init__()` method for some reason.

Solution

A bare uninitialized instance can be created by directly calling the `__new__()` method of a class. For example, consider this class:

```
class Date:
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day
```

Here's how you can create a `Date` instance without invoking `__init__()`:

```
>>> d = Date.__new__(Date)
>>> d
<__main__.Date object at 0x1006716d0>
>>> d.year
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Date' object has no attribute 'year'
>>>
```

As you can see, the resulting instance is uninitialized. Thus, it is now your responsibility to set the appropriate instance variables. For example:

```
>>> data = {'year':2012, 'month':8, 'day':29}
>>> for key, value in data.items():
...     setattr(d, key, value)
...
>>> d.year
2012
>>> d.month
8
>>>
```

Discussion

The problem of bypassing `__init__()` sometimes arises when instances are being created in a nonstandard way such as when deserializing data or in the implementation of a class method that's been defined as an alternate constructor. For example, on the `Date` class shown, someone might define an alternate constructor `today()` as follows:

```
from time import localtime

class Date:
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

    @classmethod
    def today(cls):
        d = cls.__new__(cls)
        t = localtime()
        d.year = t.tm_year
        d.month = t.tm_mon
        d.day = t.tm_mday
        return d
```

Similarly, suppose you are deserializing JSON data and, as a result, produce a dictionary like this:

```
data = { 'year': 2012, 'month': 8, 'day': 29 }
```

If you want to turn this into a `Date` instance, simply use the technique shown in the solution.

When creating instances in a nonstandard way, it's usually best to not make too many assumptions about their implementation. As such, you generally don't want to write code that directly manipulates the underlying instance dictionary `__dict__` unless you know it's guaranteed to be defined. Otherwise, the code will break if the class uses `__slots__`, properties, descriptors, or other advanced techniques. By using `setattr()` to set the values, your code will be as general purpose as possible.

Extending Classes with Mixins

Problem

You have a collection of generally useful methods that you would like to make available for extending the functionality of other class definitions. However, the classes where the methods might be added aren't necessarily related to one another via inheritance. Thus, you can't just attach the methods to a common base class.

Solution

The problem addressed by this recipe often arises in code where one is interested in the issue of class customization. For example, maybe a library provides a basic set of classes along with a set of optional customizations that can be applied if desired by the user.

To illustrate, suppose you have an interest in adding various customizations (e.g., logging, set-once, type checking, etc.) to mapping objects. Here are a set of mixin classes that do that:

```
class LoggedMappingMixin:
    '''
    Add logging to get/set/delete operations for debugging.
    '''
    __slots__ = ()

    def __getitem__(self, key):
        print('Getting ' + str(key))
        return super().__getitem__(key)

    def __setitem__(self, key, value):
        print('Setting {} = {!r}'.format(key, value))
        return super().__setitem__(key, value)

    def __delitem__(self, key):
        print('Deleting ' + str(key))
        return super().__delitem__(key)

class SetOnceMappingMixin:
    '''
    Only allow a key to be set once.
    '''
    __slots__ = ()
    def __setitem__(self, key, value):
        if key in self:
            raise KeyError(str(key) + ' already set')
        return super().__setitem__(key, value)

class StringKeysMappingMixin:
    '''
    Restrict keys to strings only
    '''
    __slots__ = ()
    def __setitem__(self, key, value):
        if not isinstance(key, str):
            raise TypeError('keys must be strings')
        return super().__setitem__(key, value)
```

These classes, by themselves, are useless. In fact, if you instantiate any one of them, it does nothing useful at all (other than generate exceptions). Instead, they are supposed to be mixed with other mapping classes through multiple inheritance. For example:

```
>>> class LoggedDict(LoggedMappingMixin, dict):
...     pass
...
>>> d = LoggedDict()
>>> d['x'] = 23
Setting x = 23
>>> d['x']
Getting x
23
>>> del d['x']
Deleting x

>>> from collections import defaultdict
>>> class SetOnceDefaultDict(SetOnceMappingMixin, defaultdict):
...     pass
...
>>> d = SetOnceDefaultDict(list)
>>> d['x'].append(2)
>>> d['y'].append(3)
>>> d['x'].append(10)
>>> d['x'] = 23
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "mixin.py", line 24, in __setitem__
    raise KeyError(str(key) + ' already set')
KeyError: 'x already set'

>>> from collections import OrderedDict
>>> class StringOrderedDict(StringKeysMappingMixin,
...                         SetOnceMappingMixin,
...                         OrderedDict):
...     pass
...
>>> d = StringOrderedDict()
>>> d['x'] = 23
>>> d[42] = 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "mixin.py", line 45, in __setitem__
    '''
TypeError: keys must be strings
>>> d['x'] = 42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "mixin.py", line 46, in __setitem__
    __slots__ = ()
  File "mixin.py", line 24, in __setitem__
    if key in self:
KeyError: 'x already set'
>>>
```

In the example, you will notice that the mixins are combined with other existing classes (e.g., `dict`, `defaultdict`, `OrderedDict`), and even one another. When combined, the classes all work together to provide the desired functionality.

Discussion

Mixin classes appear in various places in the standard library, mostly as a means for extending the functionality of other classes similar to as shown. They are also one of the main uses of multiple inheritance. For instance, if you are writing network code, you can often use the `ThreadingMixIn` from the `socketserver` module to add thread support to other network-related classes. For example, here is a multithreaded XML-RPC server:

```
from xmlrpc.server import SimpleXMLRPCServer
from socketserver import ThreadingMixIn
class ThreadedXMLRPCServer(ThreadingMixIn, SimpleXMLRPCServer):
    pass
```

It is also common to find mixins defined in large libraries and frameworks—again, typically to enhance the functionality of existing classes with optional features in some way.

There is a rich history surrounding the theory of mixin classes. However, rather than getting into all of the details, there are a few important implementation details to keep in mind.

First, mixin classes are never meant to be instantiated directly. For example, none of the classes in this recipe work by themselves. They have to be mixed with another class that implements the required mapping functionality. Similarly, the `ThreadingMixIn` from the `socketserver` library has to be mixed with an appropriate server class—it can't be used all by itself.

Second, mixin classes typically have no state of their own. This means there is no `__init__()` method and no instance variables. In this recipe, the specification of `__slots__ = ()` is meant to serve as a strong hint that the mixin classes do not have their own instance data.

If you are thinking about defining a mixin class that has an `__init__()` method and instance variables, be aware that there is significant peril associated with the fact that the class doesn't know anything about the other classes it's going to be mixed with. Thus, any instance variables created would have to be named in a way that avoids name clashes. In addition, the `__init__()` method would have to be programmed in a way that properly invokes the `__init__()` method of other classes that are mixed in. In general, this is difficult to implement since you know nothing about the argument signatures of the other classes. At the very least, you would have to implement something very general using `*arg, **kwargs`. If the `__init__()` of the mixin class took any arguments of its own, those arguments should be specified by keyword only and named in such a way to avoid name collisions with other arguments. Here is one possible implementation of a mixin defining an `__init__()` and accepting a keyword argument:

```
class RestrictKeysMixin:
    def __init__(self, *args, _restrict_key_type, **kwargs):
        self.__restrict_key_type = _restrict_key_type
        super().__init__(*args, **kwargs)

    def __setitem__(self, key, value):
        if not isinstance(key, self.__restrict_key_type):
            raise TypeError('Keys must be ' + str(self.__restrict_key_type))
        super().__setitem__(key, value)
```

Here is an example that shows how this class might be used:

```
>>> class RDict(RestrictKeysMixin, dict):
...     pass
...
>>> d = RDict(_restrict_key_type=str)
>>> e = RDict([('name','Dave'), ('n',37)], _restrict_key_type=str)
>>> f = RDict(name='Dave', n=37, _restrict_key_type=str)
>>> f
{'n': 37, 'name': 'Dave'}
>>> f[42] = 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "mixin.py", line 83, in __setitem__
    raise TypeError('Keys must be ' + str(self.__restrict_key_type))
TypeError: Keys must be <class 'str'>
>>>
```

In this example, you'll notice that initializing an `RDict()` still takes the arguments understood by `dict()`. However, there is an extra keyword argument `restrict_key_type` that is provided to the mixin class.

Finally, use of the `super()` function is an essential and critical part of writing mixin classes. In the solution, the classes redefine certain critical methods, such as `__getitem__()` and `__setitem__()`. However, they also need to call the original implementation of those methods. Using `super()` delegates to the next class on the method resolution order (MRO). This aspect of the recipe, however, is not obvious to novices, because `super()` is being used in classes that have no parent (at first glance, it might look like an error). However, in a class definition such as this:

```
class LoggedDict(LoggedMappingMixin, dict):
    pass
```

the use of `super()` in `LoggedMappingMixin` delegates to the next class over in the multiple inheritance list. That is, a call such as `super().__getitem__()` in `LoggedMappingMixin` actually steps over and invokes `dict.__getitem__()`. Without this behavior, the mixin class wouldn't work at all.

An alternative implementation of mixins involves the use of class decorators. For example, consider this code:

```
def LoggedMapping(cls):
    cls_getitem = cls.__getitem__
    cls_setitem = cls.__setitem__
    cls_delitem = cls.__delitem__

    def __getitem__(self, key):
        print('Getting ' + str(key))
        return cls_getitem(self, key)

    def __setitem__(self, key, value):
        print('Setting {} = {!r}'.format(key, value))
        return cls_setitem(self, key, value)

    def __delitem__(self, key):
        print('Deleting ' + str(key))
        return cls_delitem(self, key)

    cls.__getitem__ = __getitem__
    cls.__setitem__ = __setitem__
    cls.__delitem__ = __delitem__
    return cls
```

This function is applied as a decorator to a class definition. For example:

```
@LoggedMapping
class LoggedDict(dict):
    pass
```

If you try it, you'll find that you get the same behavior, but multiple inheritance is no longer involved. Instead, the decorator has simply performed a bit of surgery on the class definition to replace certain methods. Further details about class decorators can be found in "Using Decorators to Patch Class Definitions".

See "Implementing a Data Model or Type System" for an advanced recipe involving both mixins and class decorators.

Implementing Stateful Objects or State Machines

Problem

You want to implement a state machine or an object that operates in a number of different states, but don't want to litter your code with a lot of conditionals.

Solution

In certain applications, you might have objects that operate differently according to some kind of internal state. For example, consider a simple class representing a connection:

```
class Connection:
    def __init__(self):
        self.state = 'CLOSED'

    def read(self):
        if self.state != 'OPEN':
            raise RuntimeError('Not open')
        print('reading')

    def write(self, data):
        if self.state != 'OPEN':
           raise RuntimeError('Not open')
        print('writing')

    def open(self):
        if self.state == 'OPEN':
           raise RuntimeError('Already open')
        self.state = 'OPEN'

    def close(self):
        if self.state == 'CLOSED':
           raise RuntimeError('Already closed')
        self.state = 'CLOSED'
```

This implementation presents a couple of difficulties. First, the code is complicated by the introduction of many conditional checks for the state. Second, the performance is degraded because common operations (e.g., `read()` and `write()`) always check the state before proceeding.

A more elegant approach is to encode each operational state as a separate class and arrange for the `Connection` class to delegate to the state class. For example:

```
class Connection:
    def __init__(self):
        self.new_state(ClosedConnectionState)

    def new_state(self, newstate):
        self._state = newstate

    # Delegate to the state class
    def read(self):
        return self._state.read(self)

    def write(self, data):
        return self._state.write(self, data)

    def open(self):
        return self._state.open(self)

    def close(self):
        return self._state.close(self)

# Connection state base class
class ConnectionState:
    @staticmethod
    def read(conn):
        raise NotImplementedError()
    @staticmethod
    def write(conn, data):
        raise NotImplementedError()

    @staticmethod
    def open(conn):
        raise NotImplementedError()

    @staticmethod
    def close(conn):
        raise NotImplementedError()

# Implementation of different states
class ClosedConnectionState(ConnectionState):
    @staticmethod
    def read(conn):
        raise RuntimeError('Not open')

    @staticmethod
    def write(conn, data):
        raise RuntimeError('Not open')

    @staticmethod
    def open(conn):
        conn.new_state(OpenConnectionState)

    @staticmethod
    def close(conn):
        raise RuntimeError('Already closed')

class OpenConnectionState(ConnectionState):
    @staticmethod
    def read(conn):
        print('reading')

    @staticmethod
    def write(conn, data):
        print('writing')

    @staticmethod
    def open(conn):
        raise RuntimeError('Already open')

    @staticmethod
    def close(conn):
        conn.new_state(ClosedConnectionState)
```

Here is an interactive session that illustrates the use of these classes:

```
>>> c = Connection()
>>> c._state
<class '__main__.ClosedConnectionState'>
>>> c.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "example.py", line 10, in read
    return self._state.read(self)
  File "example.py", line 43, in read
    raise RuntimeError('Not open')
```

```
RuntimeError: Not open
>>> c.open()
>>> c._state
<class '__main__.OpenConnectionState'>
>>> c.read()
reading
>>> c.write('hello')
writing
>>> c.close()
>>> c._state
<class '__main__.ClosedConnectionState'>
>>>
```

Discussion

Writing code that features a large set of complicated conditionals and intertwined states is hard to maintain and explain. The solution presented here avoids that by splitting the individual states into their own classes.

It might look a little weird, but each state is implemented by a class with static methods, each of which take an instance of `Connection` as the first argument. This design is based on a decision to not store any instance data in the different state classes themselves. Instead, all instance data should be stored on the `Connection` instance. The grouping of states under a common base class is mostly there to help organize the code and to ensure that the proper methods get implemented. The `NotImplementedError` exception raised in base class methods is just there to make sure that subclasses provide an implementation of the required methods. As an alternative, you might consider the use of an abstract base class, as described in "Defining an Interface or Abstract Base Class".

An alternative implementation technique concerns direct manipulation of the `__class__` attribute of instances. Consider this code:

```
class Connection:
    def __init__(self):
        self.new_state(ClosedConnection)

    def new_state(self, newstate):
        self.__class__ = newstate

    def read(self):
        raise NotImplementedError()

    def write(self, data):
        raise NotImplementedError()

    def open(self):
        raise NotImplementedError()

    def close(self):
        raise NotImplementedError()

class ClosedConnection(Connection):
    def read(self):
        raise RuntimeError('Not open')

    def write(self, data):
        raise RuntimeError('Not open')
    def open(self):
        self.new_state(OpenConnection)

    def close(self):
        raise RuntimeError('Already closed')

class OpenConnection(Connection):
    def read(self):
        print('reading')

    def write(self, data):
        print('writing')

    def open(self):
        raise RuntimeError('Already open')

    def close(self):
        self.new_state(ClosedConnection)
```

The main feature of this implementation is that it eliminates an extra level of indirection. Instead of having separate `Connection` and `ConnectionState` classes, the two classes are merged together into one. As the state changes, the instance will change its type, as shown here:

```
>>> c = Connection()
>>> c
<__main__.ClosedConnection object at 0x1006718d0>
>>> c.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "state.py", line 15, in read
    raise RuntimeError('Not open')
RuntimeError: Not open
>>> c.open()
>>> c
<__main__.OpenConnection object at 0x1006718d0>
>>> c.read()
reading
>>> c.close()
>>> c
<__main__.ClosedConnection object at 0x1006718d0>
>>>
```

Object-oriented purists might be offended by the idea of simply changing the instance `__class__` attribute. However, it's technically allowed. Also, it might result in slightly faster code since all of the methods on the connection no longer involve an extra delegation step.)

Finally, either technique is useful in implementing more complicated state machines—especially in code that might otherwise feature large `if-elif-else` blocks. For example:

```
# Original implementation
class State:
    def __init__(self):
        self.state = 'A'
    def action(self, x):
        if state == 'A':
            # Action for A
            ...
            state = 'B'
        elif state == 'B':
            # Action for B
            ...
            state = 'C'
        elif state == 'C':
            # Action for C
            ...
            state = 'A'

# Alternative implementation
class State:
    def __init__(self):
        self.new_state(State_A)

    def new_state(self, state):
        self.__class__ = state

    def action(self, x):
        raise NotImplementedError()

class State_A(State):
    def action(self, x):
         # Action for A
         ...
         self.new_state(State_B)

class State_B(State):
    def action(self, x):
         # Action for B
         ...
         self.new_state(State_C)

class State_C(State):
    def action(self, x):
         # Action for C
         ...
         self.new_state(State_A)
```

This recipe is loosely based on the state design pattern found in *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley, 1995).

Calling a Method on an Object Given the Name As a String

Problem

You have the name of a method that you want to call on an object stored in a string and you want to execute the method.

Solution

For simple cases, you might use `getattr()`, like this:

```
import math

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return 'Point({!r:},{!r:})'.format(self.x, self.y)

    def distance(self, x, y):
        return math.hypot(self.x - x, self.y - y)

p = Point(2, 3)
d = getattr(p, 'distance')(0, 0)     # Calls p.distance(0, 0)
```

An alternative approach is to use `operator.methodcaller()`. For example:

```
import operator
operator.methodcaller('distance', 0, 0)(p)
```

`operator.methodcaller()` may be useful if you want to look up a method by name and supply the same arguments over and over again. For instance, if you need to sort an entire list of points:

```
points = [
    Point(1, 2),
    Point(3, 0),
    Point(10, -3),
    Point(-5, -7),
    Point(-1, 8),
    Point(3, 2)
]

# Sort by distance from origin (0, 0)
points.sort(key=operator.methodcaller('distance', 0, 0))
```

Discussion

Calling a method is actually two separate steps involving an attribute lookup and a function call. Therefore, to call a method, you simply look up the attribute using `getattr()`, as for any other attribute. To invoke the result as a method, simply treat the result of the lookup as a function.

`operator.methodcaller()` creates a callable object, but also fixes any arguments that are going to be supplied to the method. All that you need to do is provide the appropriate `self` argument. For example:

```
>>> p = Point(3, 4)
>>> d = operator.methodcaller('distance', 0, 0)
>>> d(p)
5.0
>>>
```

Invoking methods using names contained in strings is somewhat common in code that emulates case statements or variants of the visitor pattern. See the next recipe for a more advanced example.

Implementing the Visitor Pattern

Problem

You need to write code that processes or navigates through a complicated data structure consisting of many different kinds of objects, each of which needs to be handled in a different way. For example, walking through a tree structure and performing different actions depending on what kind of tree nodes are encountered.

Solution

The problem addressed by this recipe is one that often arises in programs that build data structures consisting of a large number of different kinds of objects. To illustrate, suppose you are trying to write a program that represents mathematical expressions. To do that, the program might employ a number of classes, like this:

```
class Node:
    pass

class UnaryOperator(Node):
    def __init__(self, operand):
        self.operand = operand

class BinaryOperator(Node):
    def __init__(self, left, right):
        self.left = left
        self.right = right

class Add(BinaryOperator):
    pass

class Sub(BinaryOperator):
    pass

class Mul(BinaryOperator):
    pass

class Div(BinaryOperator):
    pass

class Negate(UnaryOperator):
    pass

class Number(Node):
    def __init__(self, value):
        self.value = value
```

These classes would then be used to build up nested data structures, like this:

```
# Representation of 1 + 2 * (3 - 4) / 5
t1 = Sub(Number(3), Number(4))
t2 = Mul(Number(2), t1)
t3 = Div(t2, Number(5))
t4 = Add(Number(1), t3)
```

The problem is not the creation of such structures, but in writing code that processes them later. For example, given such an expression, a program might want to do any number of things (e.g., produce output, generate instructions, perform translation, etc.).

To enable general-purpose processing, a common solution is to implement the so-called "visitor pattern" using a class similar to this:

```
class NodeVisitor:
    def visit(self, node):
        methname = 'visit_' + type(node).__name__
        meth = getattr(self, methname, None)
        if meth is None:
            meth = self.generic_visit
        return meth(node)

    def generic_visit(self, node):
        raise RuntimeError('No {} method'.format('visit_' + type(node).__name__))
```

To use this class, a programmer inherits from it and implements various methods of the form `visit_Name()`, where `Name` is substituted with the node type. For example, if you want to evaluate the expression, you could write this:

```
class Evaluator(NodeVisitor):
    def visit_Number(self, node):
        return node.value

    def visit_Add(self, node):
        return self.visit(node.left) + self.visit(node.right)

    def visit_Sub(self, node):
        return self.visit(node.left) - self.visit(node.right)

    def visit_Mul(self, node):
        return self.visit(node.left) * self.visit(node.right)

    def visit_Div(self, node):
        return self.visit(node.left) / self.visit(node.right)

    def visit_Negate(self, node):
        return -node.operand
```

Here is an example of how you would use this class using the previously generated expression:

```
>>> e = Evaluator()
>>> e.visit(t4)
0.6
>>>
```

As a completely different example, here is a class that translates an expression into operations on a simple stack machine:

```
class StackCode(NodeVisitor):
    def generate_code(self, node):
        self.instructions = []
        self.visit(node)
        return self.instructions

    def visit_Number(self, node):
        self.instructions.append(('PUSH', node.value))

    def binop(self, node, instruction):
        self.visit(node.left)
        self.visit(node.right)
        self.instructions.append((instruction,))

    def visit_Add(self, node):
        self.binop(node, 'ADD')

    def visit_Sub(self, node):
        self.binop(node, 'SUB')

    def visit_Mul(self, node):
        self.binop(node, 'MUL')

    def visit_Div(self, node):
        self.binop(node, 'DIV')

    def unaryop(self, node, instruction):
        self.visit(node.operand)
        self.instructions.append((instruction,))

    def visit_Negate(self, node):
        self.unaryop(node, 'NEG')
```

Here is an example of this class in action:

```
>>> s = StackCode()
>>> s.generate_code(t4)
[('PUSH', 1), ('PUSH', 2), ('PUSH', 3), ('PUSH', 4), ('SUB',),
 ('MUL',), ('PUSH', 5), ('DIV',), ('ADD',)]
>>>
```

Discussion

There are really two key ideas in this recipe. The first is a design strategy where code that manipulates a complicated data structure is decoupled from the data structure itself. That is, in this recipe, none of the various `Node` classes provide any implementation that does anything with the data. Instead, all of the data manipulation is carried out by specific implementations of the separate `NodeVisitor` class. This separation makes the code extremely general purpose.

The second major idea of this recipe is in the implementation of the visitor class itself. In the visitor, you want to dispatch to a different handling method based on some value such as the node type. In a naive implementation, you might be inclined to write a huge `if` statement, like this:

```
class NodeVisitor:
    def visit(self, node):
        nodetype = type(node).__name__
        if nodetype == 'Number':
            return self.visit_Number(node)
        elif nodetype == 'Add':
            return self.visit_Add(node)
        elif nodetype == 'Sub':
            return self.visit_Sub(node)
        ...
```

However, it quickly becomes apparent that you don’t really want to take that approach. Aside from being incredibly verbose, it runs slowly, and it’s hard to maintain if you ever add or change the kind of nodes being handled. Instead, it’s much better to play a little trick where you form the name of a method and go fetch it with the `getattr()` function, as shown. The `generic_visit()` method in the solution is a fallback should no matching handler method be found. In this recipe, it raises an exception to alert the programmer that an unexpected node type was encountered.

Within each visitor class, it is common for calculations to be driven by recursive calls to the `visit()` method. For example:

```
class Evaluator(NodeVisitor):
    ...
    def visit_Add(self, node):
        return self.visit(node.left) + self.visit(node.right)
```

This recursion is what makes the visitor class traverse the entire data structure. Essentially, you keep calling `visit()` until you reach some sort of terminal node, such as `Number` in the example. The exact order of the recursion and other operations depend entirely on the application.

It should be noted that this particular technique of dispatching to a method is also a common way to emulate the behavior of switch or case statements from other languages. For example, if you are writing an HTTP framework, you might have classes that do a similar kind of dispatch:

```
class HTTPHandler:
    def handle(self, request):
        methname = 'do_' + request.request_method
        getattr(self, methname)(request)

    def do_GET(self, request):
        ...
    def do_POST(self, request):
        ...
    def do_HEAD(self, request):
        ...
```

One weakness of the visitor pattern is its heavy reliance on recursion. If you try to apply it to a deeply nested structure, it's possible that you will hit Python's recursion depth limit (see `sys.getrecursionlimit()`). To avoid this problem, you can make certain choices in your data structures. For example, you can use normal Python lists instead of linked lists or try to aggregate more data in each node to make the data more shallow. You can also try to employ nonrecursive traversal algorithms using generators or iterators as discussed in "Implementing the Visitor Pattern Without Recursion".

Use of the visitor pattern is extremely common in programs related to parsing and compiling. One notable implementation can be found in Python's own `ast` module. In addition to allowing traversal of tree structures, it provides a variation that allows a data structure to be rewritten or transformed as it is traversed (e.g., nodes added or removed). Look at the source for `ast` for more details. "Parsing and Analyzing Python Source" shows an example of using the `ast` module to process Python source code.

Implementing the Visitor Pattern Without Recursion

Problem

You're writing code that navigates through a deeply nested tree structure using the visitor pattern, but it blows up due to exceeding the recursion limit. You'd like to eliminate the recursion, but keep the programming style of the visitor pattern.

Solution

Clever use of generators can sometimes be used to eliminate recursion from algorithms involving tree traversal or searching. In “Implementing the Visitor Pattern”, a visitor class was presented. Here is an alternative implementation of that class that drives the computation in an entirely different way using a stack and generators:

```
import types

class Node:
    pass

import types
class NodeVisitor:
    def visit(self, node):
        stack = [ node ]
        last_result = None
        while stack:
            try:
                last = stack[-1]
                if isinstance(last, types.GeneratorType):
                    stack.append(last.send(last_result))
                    last_result = None
                elif isinstance(last, Node):
                    stack.append(self._visit(stack.pop()))
                else:
                    last_result = stack.pop()
            except StopIteration:
                stack.pop()
        return last_result

    def _visit(self, node):
        methname = 'visit_' + type(node).__name__
        meth = getattr(self, methname, None)
        if meth is None:
            meth = self.generic_visit
        return meth(node)

    def generic_visit(self, node):
        raise RuntimeError('No {} method'.format('visit_' + type(node).__name__))
```

If you use this class, you’ll find that it still works with existing code that might have used recursion. In fact, you can use it as a drop-in replacement for the visitor implementation in the prior recipe. For example, consider the following code, which involves expression trees:

```
class UnaryOperator(Node):
    def __init__(self, operand):
        self.operand = operand

class BinaryOperator(Node):
    def __init__(self, left, right):
        self.left = left
        self.right = right

class Add(BinaryOperator):
    pass

class Sub(BinaryOperator):
    pass

class Mul(BinaryOperator):
    pass
```

```
class Div(BinaryOperator):
    pass

class Negate(UnaryOperator):
    pass

class Number(Node):
    def __init__(self, value):
        self.value = value

# A sample visitor class that evaluates expressions
class Evaluator(NodeVisitor):
    def visit_Number(self, node):
        return node.value

    def visit_Add(self, node):
        return self.visit(node.left) + self.visit(node.right)

    def visit_Sub(self, node):
        return self.visit(node.left) - self.visit(node.right)

    def visit_Mul(self, node):
        return self.visit(node.left) * self.visit(node.right)

    def visit_Div(self, node):
        return self.visit(node.left) / self.visit(node.right)

    def visit_Negate(self, node):
        return -self.visit(node.operand)

if __name__ == '__main__':
    # 1 + 2*(3-4) / 5
    t1 = Sub(Number(3), Number(4))
    t2 = Mul(Number(2), t1)
    t3 = Div(t2, Number(5))
    t4 = Add(Number(1), t3)

    # Evaluate it
    e = Evaluator()
    print(e.visit(t4))     # Outputs 0.6
```

The preceding code works for simple expressions. However, the implementation of `Evaluator` uses recursion and crashes if things get too nested. For example:

```
>>> a = Number(0)
>>> for n in range(1, 100000):
...     a = Add(a, Number(n))
...
>>> e = Evaluator()
>>> e.visit(a)
Traceback (most recent call last):
...
  File "visitor.py", line 29, in _visit
    return meth(node)
  File "visitor.py", line 67, in visit_Add
    return self.visit(node.left) + self.visit(node.right)
RuntimeError: maximum recursion depth exceeded
>>>
```

Now let’s change the `Evaluator` class ever so slightly to the following:

```
class Evaluator(NodeVisitor):
    def visit_Number(self, node):
        return node.value

    def visit_Add(self, node):
        yield (yield node.left) + (yield node.right)

    def visit_Sub(self, node):
        yield (yield node.left) - (yield node.right)

    def visit_Mul(self, node):
        yield (yield node.left) * (yield node.right)

    def visit_Div(self, node):
        yield (yield node.left) / (yield node.right)

    def visit_Negate(self, node):
        yield -(yield node.operand)
```

If you try the same recursive experiment, you’ll find that it suddenly works. It’s magic!

```
>>> a = Number(0)
>>> for n in range(1,100000):
...     a = Add(a, Number(n))
...
>>> e = Evaluator()
>>> e.visit(a)
4999950000
>>>
```

If you want to add custom processing into any of the methods, it still works. For example:

```
class Evaluator(NodeVisitor):
    ...
    def visit_Add(self, node):
        print('Add:', node)
        lhs = yield node.left
        print('left=', lhs)
        rhs = yield node.right
        print('right=', rhs)
        yield lhs + rhs
    ...
```

Here is some sample output:

```
>>> e = Evaluator()
>>> e.visit(t4)
Add: <__main__.Add object at 0x1006a8d90>
left= 1
right= -0.4
0.6
>>>
```

Discussion

This recipe nicely illustrates how generators and coroutines can perform mind-bending tricks involving program control flow, often to great advantage. To understand this recipe, a few key insights are required.

First, in problems related to tree traversal, a common implementation strategy for avoiding recursion is to write algorithms involving a stack or queue. For example, depth-first traversal can be implemented entirely by pushing nodes onto a stack when first encountered and then popping them off once processing has finished. The central core of the `visit()` method in the solution is built around this idea. The algorithm starts by pushing the initial node onto the `stack` list and runs until the stack is empty. During execution, the stack will grow according to the depth of the underlying tree structure.

The second insight concerns the behavior of the `yield` statement in generators. When `yield` is encountered, the behavior of a generator is to emit a value and to suspend. This recipe uses this as a replacement for recursion. For example, instead of writing a recursive expression like this:

```
value = self.visit(node.left)
```

you replace it with the following:

```
value = yield node.left
```

Behind the scenes, this sends the node in question (`node.left`) back to the `visit()` method. The `visit()` method then carries out the execution of the appropriate `visit_Name()` method for that node. In some sense, this is almost the opposite of recursion. That is, instead of calling `visit()` recursively to move the algorithm forward, the `yield` statement is being used to temporarily back out of the computation in progress. Thus, the `yield` is essentially a signal that tells the algorithm that the yielded node needs to be processed first before further progress can be made.

The final part of this recipe concerns propagation of results. When generator functions are used, you can no longer use `return` statements to emit values (doing so will cause a `SyntaxError` exception). Thus, the `yield` statement has to do double duty to cover the case. In this recipe, if the value produced by a `yield` statement is a non-Node type, it is assumed to be a value that will be propagated to the next step of the calculation. This is the purpose of the `last_return` variable in the code. Typically, this would hold the last value yielded by a visit method. That value would then be sent into the previously executing method, where it would show up as the return value from a `yield` statement. For example, in this code:

```
value = yield node.left
```

The `value` variable gets the value of `last_return`, which is the result returned by the visitor method invoked for `node.left`.

All of these aspects of the recipe are found in this fragment of code:

```
try:
    last = stack[-1]
    if isinstance(last, types.GeneratorType):
        stack.append(last.send(last_result))
        last_result = None
    elif isinstance(last, Node):
        stack.append(self._visit(stack.pop()))
    else:
        last_result = stack.pop()
except StopIteration:
    stack.pop()
```

The code works by simply looking at the top of the stack and deciding what to do next. If it’s a generator, then its `send()` method is invoked with the last result (if any) and the result appended onto the stack for further processing. The value returned by `send()` is the same value that was given to the `yield` statement. Thus, in a statement such as `yield node.left`, the `Node` instance `node.left` is returned by `send()` and placed on the top of the stack.

If the top of the stack is a `Node` instance, then it is replaced by the result of calling the appropriate visit method for that node. This is where the underlying recursion is being eliminated. Instead of the various visit methods directly calling `visit()` recursively, it takes place here. As long as the methods use `yield`, it all works out.

Finally, if the top of the stack is anything else, it's assumed to be a return value of some kind. It just gets popped off the stack and placed into `last_result`. If the next item on the stack is a generator, then it gets sent in as a return value for the `yield`. It should be noted that the final return value of `visit()` is also set to `last_result`. This is what makes this recipe work with a traditional recursive implementation. If no generators are being used, this value simply holds the value given to any `return` statements used in the code.

One potential danger of this recipe concerns the distinction between yielding `Node` and non-`Node` values. In the implementation, all `Node` instances are automatically traversed. This means that you can't use a `Node` as a return value to be propagated. In practice, this may not matter. However, if it does, you might need to adapt the algorithm slightly. For example, possibly by introducing another class into the mix, like this:

```
class Visit:
    def __init__(self, node):
        self.node = node

class NodeVisitor:
    def visit(self, node):
        stack = [ Visit(node) ]
        last_result = None
        while stack:
            try:
                last = stack[-1]
                if isinstance(last, types.GeneratorType):
                    stack.append(last.send(last_result))
                    last_result = None
                elif isinstance(last, Visit):
                    stack.append(self._visit(stack.pop().node))
                else:
                    last_result = stack.pop()
            except StopIteration:
                stack.pop()
        return last_result

    def _visit(self, node):
        methname = 'visit_' + type(node).__name__
        meth = getattr(self, methname, None)
        if meth is None:
            meth = self.generic_visit
        return meth(node)

    def generic_visit(self, node):
        raise RuntimeError('No {} method'.format('visit_' + type(node).__name__))
```

With this implementation, the various visitor methods would now look like this:

```
class Evaluator(NodeVisitor):
    ...
    def visit_Add(self, node):
        yield (yield Visit(node.left)) + (yield Visit(node.right))

    def visit_Sub(self, node):
        yield (yield Visit(node.left)) - (yield Visit(node.right))
    ...
```

Having seen this recipe, you might be inclined to investigate a solution that doesn't involve `yield`. However, doing so will lead to code that has to deal with many of the same issues presented here. For example, to eliminate recursion, you'll need to maintain a stack. You'll also need to come up with some scheme for managing the traversal and invoking various visitor-related logic. Without generators, this code ends up being a very messy mix of stack manipulation, callback functions, and other constructs. Frankly, the main benefit of using `yield` is that you can write nonrecursive code in an elegant style that looks almost exactly like the recursive implementation.

Managing Memory in Cyclic Data Structures

Problem

Your program creates data structures with cycles (e.g., trees, graphs, observer patterns, etc.), but you are experiencing problems with memory management.

Solution

A simple example of a cyclic data structure is a tree structure where a parent points to its children and the children point back to their parent. For code like this, you should consider making one of the links a weak reference using the `weakref` library. For example:

```
import weakref

class Node:
    def __init__(self, value):
        self.value = value
        self._parent = None
        self.children = []

    def __repr__(self):
        return 'Node({!r:})'.format(self.value)

    # property that manages the parent as a weak-reference
    @property
    def parent(self):
        return self._parent if self._parent is None else self._parent()

    @parent.setter
    def parent(self, node):
        self._parent = weakref.ref(node)

    def add_child(self, child):
        self.children.append(child)
        child.parent = self
```

This implementation allows the parent to quietly die. For example:

```
>>> root = Node('parent')
>>> c1 = Node('child')
>>> root.add_child(c1)
>>> print(c1.parent)
Node('parent')
>>> del root
>>> print(c1.parent)
None
>>>
```

Discussion

Cyclic data structures are a somewhat tricky aspect of Python that require careful study because the usual rules of garbage collection often don’t apply. For example, consider this code:

```
# Class just to illustrate when deletion occurs
class Data:
    def __del__(self):
        print('Data.__del__')

# Node class involving a cycle
class Node:
    def __init__(self):
        self.data = Data()
        self.parent = None
        self.children = []
    def add_child(self, child):
        self.children.append(child)
        child.parent = self
```

Now, using this code, try some experiments to see some subtle issues with garbage collection:

```
>>> a = Data()
>>> del a               # Immediately deleted
Data.__del__
>>> a = Node()
>>> del a               # Immediately deleted
Data.__del__
>>> a = Node()
>>> a.add_child(Node())
>>> del a               # Not deleted (no message)
>>>
```

As you can see, objects are deleted immediately all except for the last case involving a cycle. The reason is that Python’s garbage collection is based on simple reference counting. When the reference count of an object reaches 0, it is immediately deleted. For cyclic data structures, however, this never happens. Thus, in the last part of the example, the parent and child nodes refer to each other, keeping the reference count nonzero.

To deal with cycles, there is a separate garbage collector that runs periodically. However, as a general rule, you never know when it might run. Consequently, you never really know when cyclic data structures might get collected. If necessary, you can force garbage collection, but doing so is a bit clunky:

```
>>> import gc
>>> gc.collect()     # Force collection
Data.__del__
Data.__del__
```

```
>>>
```

An even worse problem occurs if the objects involved in a cycle define their own `__del__()` method. For example, suppose the code looked like this:

```
# Class just to illustrate when deletion occurs
class Data:
    def __del__(self):
        print('Data.__del__')

# Node class involving a cycle
class Node:
    def __init__(self):
        self.data = Data()
        self.parent = None
        self.children = []

    # NEVER DEFINE LIKE THIS.
    # Only here to illustrate pathological behavior
    def __del__(self):
        del self.data
        del.parent
        del.children

    def add_child(self, child):
        self.children.append(child)
        child.parent = self
```

In this case, the data structures will never be garbage collected at all and your program will leak memory! If you try it, you'll see that the `Data.__del__` message never appears at all—even after a forced garbage collection:

```
>>> a = Node()
>>> a.add_child(Node()
>>> del a             # No message (not collected)
>>> import gc
>>> gc.collect()      # No message (not collected)
>>>
```

Weak references solve this problem by eliminating reference cycles. Essentially, a weak reference is a pointer to an object that does not increase its reference count. You create weak references using the `weakref` library. For example:

```
>>> import weakref
>>> a = Node()
>>> a_ref = weakref.ref(a)
>>> a_ref
<weakref at 0x100581f70; to 'Node' at 0x1005c5410>
>>>
```

To dereference a weak reference, you call it like a function. If the referenced object still exists, it is returned. Otherwise, `None` is returned. Since the reference count of the original object wasn't increased, it can be deleted normally. For example:

```
>>> print(a_ref())
<__main__.Node object at 0x1005c5410>
>>> del a
Data.__del__
```

```
>>> print(a_ref())
None
>>>
```

By using weak references, as shown in the solution, you'll find that there are no longer any reference cycles and that garbage collection occurs immediately once a node is no longer being used. See "Creating Cached Instances" for another example involving weak references.

Making Classes Support Comparison Operations

Problem

You'd like to be able to compare instances of your class using the standard comparison operators (e.g., `>=`, `!=`, `<=`, etc.), but without having to write a lot of special methods.

Solution

Python classes can support comparison by implementing a special method for each comparison operator. For example, to support the `>=` operator, you define a `__ge__()` method in the classes. Although defining a single method is usually no problem, it quickly gets tedious to create implementations of every possible comparison operator.

The `functools.total_ordering` decorator can be used to simplify this process. To use it, you decorate a class with it, and define `__eq__()` and one other comparison method (`__lt__`, `__le__`, `__gt__`, or `__ge__`). The decorator then fills in the other comparison methods for you.

As an example, let's build some houses and add some rooms to them, and then perform comparisons based on the size of the houses:

```
from functools import total_ordering
class Room:
    def __init__(self, name, length, width):
        self.name = name
        self.length = length
        self.width = width
        self.square_feet = self.length * self.width

@total_ordering
class House:
    def __init__(self, name, style):
        self.name = name
        self.style = style
        self.rooms = list()

    @property
    def living_space_footage(self):
        return sum(r.square_feet for r in self.rooms)

    def add_room(self, room):
        self.rooms.append(room)

    def __str__(self):
        return '{}: {} square foot {}'.format(self.name,
                                              self.living_space_footage,
                                                  self.style)

    def __eq__(self, other):
        return self.living_space_footage == other.living_space_footage

    def __lt__(self, other):
        return self.living_space_footage < other.living_space_footage
```

Here, the `House` class has been decorated with `@total_ordering`. Definitions of `__eq__()` and `__lt__()` are provided to compare houses based on the total square footage of their rooms. This minimum definition is all that is required to make all of the other comparison operations work. For example:

```
# Build a few houses, and add rooms to them
h1 = House('h1', 'Cape')
h1.add_room(Room('Master Bedroom', 14, 21))
h1.add_room(Room('Living Room', 18, 20))
h1.add_room(Room('Kitchen', 12, 16))
h1.add_room(Room('Office', 12, 12))

h2 = House('h2', 'Ranch')
h2.add_room(Room('Master Bedroom', 14, 21))
h2.add_room(Room('Living Room', 18, 20))
h2.add_room(Room('Kitchen', 12, 16))

h3 = House('h3', 'Split')
h3.add_room(Room('Master Bedroom', 14, 21))
h3.add_room(Room('Living Room', 18, 20))
h3.add_room(Room('Office', 12, 16))
h3.add_room(Room('Kitchen', 15, 17))
houses = [h1, h2, h3]

print('Is h1 bigger than h2?', h1 > h2) # prints True
print('Is h2 smaller than h3?', h2 < h3) # prints True
print('Is h2 greater than or equal to h1?', h2 >= h1) # Prints False
print('Which one is biggest?', max(houses)) # Prints 'h3: 1101-square-foot Split'
print('Which is smallest?', min(houses)) # Prints 'h2: 846-square-foot Ranch'
```

Discussion

If you've written the code to make a class support all of the basic comparison operators, then `total_ordering` probably doesn't seem all that magical: it literally defines a mapping from each of the comparison-supporting methods to all of the other ones that would be required. So, if you defined `__lt__()` in your class as in the solution, it is used to build all of the other comparison operators. It's really just filling in the class with methods like this:

```
class House:
    def __eq__(self, other):
        ...
    def __lt__(self, other):
        ...

    # Methods created by @total_ordering
    __le__ = lambda self, other: self < other or self == other
    __gt__ = lambda self, other: not (self < other or self == other)
    __ge__ = lambda self, other: not (self < other)
    __ne__ = lambda self, other: not self == other
```

Sure, it's not hard to write these methods yourself, but `@total_ordering` simply takes the guesswork out of it.

Creating Cached Instances

Problem

When creating instances of a class, you want to return a cached reference to a previous instance created with the same arguments (if any).

Solution

The problem being addressed in this recipe sometimes arises when you want to ensure that there is only one instance of a class created for a set of input arguments. Practical examples include the behavior of libraries, such as the `logging` module, that only want to associate a single logger instance with a given name. For example:

```
>>> import logging
>>> a = logging.getLogger('foo')
>>> b = logging.getLogger('bar')
>>> a is b
False
>>> c = logging.getLogger('foo')
>>> a is c
True
>>>
```

To implement this behavior, you should make use of a factory function that's separate from the class itself. For example:

```
# The class in question
class Spam:
    def __init__(self, name):
        self.name = name

# Caching support
import weakref
_spam_cache = weakref.WeakValueDictionary()

def get_spam(name):
    if name not in _spam_cache:
        s = Spam(name)
        _spam_cache[name] = s
    else:
        s = _spam_cache[name]
    return s
```

If you use this implementation, you'll find that it behaves in the manner shown earlier:

```
>>> a = get_spam('foo')
>>> b = get_spam('bar')
>>> a is b
False
>>> c = get_spam('foo')
>>> a is c
True
```

```
>>>
```

Discussion

Writing a special factory function is often a simple approach for altering the normal rules of instance creation. One question that often arises at this point is whether or not a more elegant approach could be taken.

For example, you might consider a solution that redefines the `__new__()` method of a class as follows:

```
# Note: This code doesn't quite work
import weakref

class Spam:
    _spam_cache = weakref.WeakValueDictionary()
    def __new__(cls, name):
        if name in cls._spam_cache:
            return cls._spam_cache[name]
        else:
            self = super().__new__(cls)
            cls._spam_cache[name] = self
            return self

    def __init__(self, name):
        print('Initializing Spam')
        self.name = name
```

At first glance, it seems like this code might do the job. However, a major problem is that the `__init__()` method always gets called, regardless of whether the instance was cached or not. For example:

```
>>> s = Spam('Dave')
Initializing Spam
>>> t = Spam('Dave')
Initializing Spam
>>> s is t
True
>>>
```

That behavior is probably not what you want. So, to solve the problem of caching without reinitialization, you need to take a slightly different approach.

The use of weak references in this recipe serves an important purpose related to garbage collection, as described in "Managing Memory in Cyclic Data Structures". When maintaining a cache of instances, you often only want to keep items in the cache as long as they're actually being used somewhere in the program. A `WeakValueDictionary` instance only holds onto the referenced items as long as they exist somewhere else. Otherwise, the dictionary keys disappear when instances are no longer being used. Observe:

```
>>> a = get_spam('foo')
>>> b = get_spam('bar')
>>> c = get_spam('foo')
>>> list(_spam_cache)
['foo', 'bar']
>>> del a
>>> del c
>>> list(_spam_cache)
```

```
['bar']
>>> del b
>>> list(_spam_cache)
[]
>>>
```

For many programs, the bare-bones code shown in this recipe will often suffice. However, there are a number of more advanced implementation techniques that can be considered.

One immediate concern with this recipe might be its reliance on global variables and a factory function that's decoupled from the original class definition. One way to clean this up is to put the caching code into a separate manager class and glue things together like this:

```
import weakref

class CachedSpamManager:
    def __init__(self):
        self._cache = weakref.WeakValueDictionary()
    def get_spam(self, name):
        if name not in self._cache:
            s = Spam(name)
            self._cache[name] = s
        else:
            s = self._cache[name]
        return s

    def clear(self):
        self._cache.clear()

class Spam:
    manager = CachedSpamManager()
    def __init__(self, name):
        self.name = name

def get_spam(name):
    return Spam.manager.get_spam(name)
```

One feature of this approach is that it affords a greater degree of potential flexibility. For example, different kinds of management schemes could be be implemented (as separate classes) and attached to the `Spam` class as a replacement for the default caching implementation. None of the other code (e.g., `get_spam`) would need to be changed to make it work.

Another design consideration is whether or not you want to leave the class definition exposed to the user. If you do nothing, a user can easily make instances, bypassing the caching mechanism:

```
>>> a = Spam('foo')
>>> b = Spam('foo')
>>> a is b
False
>>>
```

If preventing this is important, you can take certain steps to avoid it. For example, you might give the class a name starting with an underscore, such as `_Spam`, which at least gives the user a clue that they shouldn't access it directly.

Alternatively, if you want to give users a stronger hint that they shouldn't instantiate `Spam` instances directly, you can make `__init__()` raise an exception and use a class method to make an alternate constructor like this:

```
class Spam:
    def __init__(self, *args, **kwargs):
        raise RuntimeError("Can't instantiate directly")

    # Alternate constructor
    @classmethod
    def _new(cls, name):
        self = cls.__new__(cls)
        self.name = name
```

To use this, you modify the caching code to use `Spam._new()` to create instances instead of the usual call to `Spam()`. For example:

```
import weakref

class CachedSpamManager:
    def __init__(self):
        self._cache = weakref.WeakValueDictionary()
    def get_spam(self, name):
        if name not in self._cache:
            s = Spam._new(name)          # Modified creation
            self._cache[name] = s
        else:
            s = self._cache[name]
        return s
```

Although there are more extreme measures that can be taken to hide the visibility of the `Spam` class, it's probably best to not overthink the problem. Using an underscore on the name or defining a class method constructor is usually enough for programmers to get a hint.

Caching and other creational patterns can often be solved in a more elegant (albeit advanced) manner through the use of metaclasses. See "Using a Metaclass to Control Instance Creation".

Prev Next
Chapter 7. Functions Home Chapter 9. Metaprogramming