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