Chapter 9. Metaprogramming

Prev Next

Chapter 9. Metaprogramming

One of the most important mantras of software development is "don't repeat yourself." That is, any time you are faced with a problem of creating highly repetitive code (or cutting or pasting source code), it often pays to look for a more elegant solution. In Python, such problems are often solved under the category of "metaprogramming." In a nutshell, metaprogramming is about creating functions and classes whose main goal is to manipulate code (e.g., modifying, generating, or wrapping existing code). The main features for this include decorators, class decorators, and metaclasses. However, a variety of other useful topics—including signature objects, execution of code with `exec()`, and inspecting the internals of functions and classes—enter the picture. The main purpose of this chapter is to explore various metaprogramming

techniques and to give examples of how they can be used to customize the behavior of Python to your own whims.

Putting a Wrapper Around a Function

Problem

You want to put a wrapper layer around a function that adds extra processing (e.g., logging, timing, etc.).

Solution

If you ever need to wrap a function with extra code, define a decorator function. For example:

```
import time
from functools import wraps

def timethis(func):
    '''
    Decorator that reports the execution time.
    '''
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(func.__name__, end-start)
        return result
    return wrapper
```

Here is an example of using the decorator:

```
>>> @timethis
... def countdown(n):
...     '''
...     Counts down
...     '''
...     while n > 0:
...             n -= 1
...
>>> countdown(100000)
countdown 0.008917808532714844
>>> countdown(10000000)
countdown 0.87188299392912
>>>
```

Discussion

A decorator is a function that accepts a function as input and returns a new function as output. Whenever you write code like this:

```
@timethis
def countdown(n):
    ...
```

it's the same as if you had performed these separate steps:

```
def countdown(n):
    ...
countdown = timethis(countdown)
```

As an aside, built-in decorators such as `@staticmethod`, `@classmethod`, and `@property` work in the same way. For example, these two code fragments are equivalent:

```
class A:
    @classmethod
    def method(cls):
        pass

class B:
    # Equivalent definition of a class method
    def method(cls):
        pass
    method = classmethod(method)
```

The code inside a decorator typically involves creating a new function that accepts any arguments using `*args` and `**kwargs`, as shown with the `wrapper()` function in this recipe. Inside this function, you place a call to the original input function and return its result. However, you also place whatever extra code you want to add (e.g., timing). The newly created function `wrapper` is returned as a result and takes the place of the original function.

It's critical to emphasize that decorators generally do not alter the calling signature or return value of the function being wrapped. The use of `*args` and `**kwargs` is there to make sure that any input arguments can be accepted. The return value of a decorator is almost always the result of calling `func(*args, **kwargs)`, where `func` is the original unwrapped function.

When first learning about decorators, it is usually very easy to get started with some simple examples, such as the one shown. However, if you are going to write decorators for real, there are some subtle details to consider. For example, the use of the decorator `@wraps(func)` in the solution is an easy to forget but important technicality related to preserving function metadata, which is described in the next recipe. The next few recipes that follow fill in some details that will be important if you wish to write decorator functions of your own.

Preserving Function Metadata When Writing Decorators

Problem

You've written a decorator, but when you apply it to a function, important metadata such as the name, doc string, annotations, and calling signature are lost.

Solution

Whenever you define a decorator, you should always remember to apply the `@wraps` decorator from the `functools` library to the underlying wrapper function. For example:

```
import time
from functools import wraps

def timethis(func):
    '''
```

```
    Decorator that reports the execution time.
    '''
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(func.__name__, end-start)
        return result
    return wrapper
```

Here is an example of using the decorator and examining the resulting function metadata:

```
>>> @timethis
... def countdown(n:int):
...     '''
...     Counts down
...     '''
...     while n > 0:
...             n -= 1
...
>>> countdown(100000)
countdown 0.008917808532714844
>>> countdown.__name__
'countdown'
>>> countdown.__doc__
'\n\tCounts down\n\t'
>>> countdown.__annotations__
{'n': <class 'int'>}
>>>
```

Discussion

Copying decorator metadata is an important part of writing decorators. If you forget to use `@wraps`, you'll find that the decorated function loses all sorts of useful information. For instance, if omitted, the metadata in the last example would look like this:

```
>>> countdown.__name__
'wrapper'
>>> countdown.__doc__
>>> countdown.__annotations__
{}
>>>
```

An important feature of the `@wraps` decorator is that it makes the wrapped function available to you in the `__wrapped__` attribute. For example, if you want to access the wrapped function directly, you could do this:

```
>>> countdown.__wrapped__(100000)
>>>
```

The presence of the `__wrapped__` attribute also makes decorated functions properly expose the underlying signature of the wrapped function. For example:

```
>>> from inspect import signature
>>> print(signature(countdown))
(n:int)
```

```
>>>
```

One common question that sometimes arises is how to make a decorator that directly copies the calling signature of the original function being wrapped (as opposed to using `*args` and `**kwargs`). In general, this is difficult to implement without resorting to some trick involving the generator of code strings and `exec()`. Frankly, you're usually best off using `@wraps` and relying on the fact that the underlying function signature can be propagated by access to the underlying `__wrapped__` attribute. See "Enforcing an Argument Signature on *args and **kwargs" for more information about signatures.

Unwrapping a Decorator

Problem

A decorator has been applied to a function, but you want to "undo" it, gaining access to the original unwrapped function.

Solution

Assuming that the decorator has been implemented properly using `@wraps` (see "Preserving Function Metadata When Writing Decorators"), you can usually gain access to the original function by accessing the `__wrapped__` attribute. For example:

```
>>> @somedecorator
>>> def add(x, y):
...     return x + y
...
>>> orig_add = add.__wrapped__
>>> orig_add(3, 4)
7
>>>
```

Discussion

Gaining direct access to the unwrapped function behind a decorator can be useful for debugging, introspection, and other operations involving functions. However, this recipe only works if the implementation of a decorator properly copies metadata using `@wraps` from the `functools` module or sets the `__wrapped__` attribute directly.

If multiple decorators have been applied to a function, the behavior of accessing `__wrapped__` is currently undefined and should probably be avoided. In Python 3.3, it bypasses all of the layers. For example, suppose you have code like this:

```
from functools import wraps

def decorator1(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print('Decorator 1')
        return func(*args, **kwargs)
    return wrapper

def decorator2(func):
    @wraps(func)
```

```
    def wrapper(*args, **kwargs):
        print('Decorator 2')
        return func(*args, **kwargs)
    return wrapper

@decorator1
@decorator2
def add(x, y):
    return x + y
```

Here is what happens when you call the decorated function and the original function through `__wrapped__`:

```
>>> add(2, 3)
Decorator 1
Decorator 2
5
>>> add.__wrapped__(2, 3)
5
>>>
```

However, this behavior has been reported as a bug (see http://bugs.python.org/issue17482) and may be changed to explose the proper decorator chain in a future release.

Last, but not least, be aware that not all decorators utilize `@wraps`, and thus, they may not work as described. In particular, the built-in decorators `@staticmethod` and `@classmethod` create descriptor objects that don't follow this convention (instead, they store the original function in a `__func__` attribute). Your mileage may vary.

Defining a Decorator That Takes Arguments

Problem

You want to write a decorator function that takes arguments.

Solution

Let's illustrate the process of accepting arguments with an example. Suppose you want to write a decorator that adds logging to a function, but allows the user to specify the logging level and other details as arguments. Here is how you might define the decorator:

```
from functools import wraps
import logging

def logged(level, name=None, message=None):
    '''
    Add logging to a function.  level is the logging
    level, name is the logger name, and message is the
    log message.  If name and message aren't specified,
    they default to the function's module and name.
    '''
    def decorate(func):
        logname = name if name else func.__module__
        log = logging.getLogger(logname)
        logmsg = message if message else func.__name__
```

```
        @wraps(func)
        def wrapper(*args, **kwargs):
            log.log(level, logmsg)
            return func(*args, **kwargs)
        return wrapper
    return decorate

# Example use
@logged(logging.DEBUG)
def add(x, y):
    return x + y

@logged(logging.CRITICAL, 'example')
def spam():
    print('Spam!')
```

On first glance, the implementation looks tricky, but the idea is relatively simple. The outermost function `logged()` accepts the desired arguments and simply makes them available to the inner functions of the decorator. The inner function `decorate()` accepts a function and puts a wrapper around it as normal. The key part is that the wrapper is allowed to use the arguments passed to `logged()`.

Discussion

Writing a decorator that takes arguments is tricky because of the underlying calling sequence involved. Specifically, if you have code like this:

```
@decorator(x, y, z)
def func(a, b):
    pass
```

The decoration process evaluates as follows:

```
def func(a, b):
    pass

func = decorator(x, y, z)(func)
```

Carefully observe that the result of `decorator(x, y, z)` must be a callable which, in turn, takes a function as input and wraps it. See "Enforcing Type Checking on a Function Using a Decorator" for another example of a decorator taking arguments.

Defining a Decorator with User Adjustable Attributes

Problem

You want to write a decorator function that wraps a function, but has user adjustable attributes that can be used to control the behavior of the decorator at runtime.

Solution

Here is a solution that expands on the last recipe by introducing accessor functions that change internal variables through the use of `nonlocal` variable declarations. The accessor functions are then attached to

the wrapper function as function attributes.

```
from functools import wraps, partial
import logging

# Utility decorator to attach a function as an attribute of obj
def attach_wrapper(obj, func=None):
    if func is None:
        return partial(attach_wrapper, obj)
    setattr(obj, func.__name__, func)
    return func

def logged(level, name=None, message=None):
    '''
    Add logging to a function.  level is the logging
    level, name is the logger name, and message is the
    log message.  If name and message aren't specified,
    they default to the function's module and name.
    '''
    def decorate(func):
        logname = name if name else func.__module__
        log = logging.getLogger(logname)
        logmsg = message if message else func.__name__

        @wraps(func)
        def wrapper(*args, **kwargs):
            log.log(level, logmsg)
            return func(*args, **kwargs)

        # Attach setter functions
        @attach_wrapper(wrapper)
        def set_level(newlevel):
            nonlocal level
            level = newlevel

        @attach_wrapper(wrapper)
        def set_message(newmsg):
            nonlocal logmsg
            logmsg = newmsg

        return wrapper
    return decorate

# Example use
@logged(logging.DEBUG)
def add(x, y):
    return x + y

@logged(logging.CRITICAL, 'example')
def spam():
    print('Spam!')
```

Here is an interactive session that shows the various attributes being changed after definition:

```
>>> import logging
>>> logging.basicConfig(level=logging.DEBUG)
>>> add(2, 3)
DEBUG:__main__:add
5
```

```
>>> # Change the log message
>>> add.set_message('Add called')
>>> add(2, 3)
DEBUG:__main__:Add called
5

>>> # Change the log level
>>> add.set_level(logging.WARNING)
>>> add(2, 3)
WARNING:__main__:Add called
5
>>>
```

Discussion

The key to this recipe lies in the accessor functions [e.g., `set_message()` and `set_level()`] that get attached to the wrapper as attributes. Each of these accessors allows internal parameters to be adjusted through the use of `nonlocal` assignments.

An amazing feature of this recipe is that the accessor functions will propagate through multiple levels of decoration (if all of your decorators utilize `@functools.wraps`). For example, suppose you introduced an additional decorator, such as the `@timethis` decorator from "Preserving Function Metadata When Writing Decorators", and wrote code like this:

```
@timethis
@logged(logging.DEBUG)
def countdown(n):
    while n > 0:
        n -= 1
```

You'll find that the accessor methods still work:

```
>>> countdown(10000000)
DEBUG:__main__:countdown
countdown 0.8198461532592773
>>> countdown.set_level(logging.WARNING)
>>> countdown.set_message("Counting down to zero")
>>> countdown(10000000)
WARNING:__main__:Counting down to zero
countdown 0.8225970268249512
>>>
```

You'll also find that it all still works exactly the same way if the decorators are composed in the opposite order, like this:

```
@logged(logging.DEBUG)
@timethis
def countdown(n):
    while n > 0:
        n -= 1
```

Although it's not shown, accessor functions to return the value of various settings could also be written just as easily by adding extra code such as this:

```
...
@attach_wrapper(wrapper)
def get_level():
```

```
    return level

# Alternative
wrapper.get_level = lambda: level
...
```

One extremely subtle facet of this recipe is the choice to use accessor functions in the first place. For example, you might consider an alternative formulation solely based on direct access to function attributes like this:

```
...
@wraps(func)
def wrapper(*args, **kwargs):
    wrapper.log.log(wrapper.level, wrapper.logmsg)
    return func(*args, **kwargs)

# Attach adjustable attributes
wrapper.level = level
wrapper.logmsg = logmsg
wrapper.log = log
...
```

This approach would work to a point, but only if it was the topmost decorator. If you had another decorator applied on top (such as the `@timethis` example), it would shadow the underlying attributes and make them unavailable for modification. The use of accessor functions avoids this limitation.

Last, but not least, the solution shown in this recipe might be a possible alternative for decorators defined as classes, as shown in "Defining Decorators As Classes".

Defining a Decorator That Takes an Optional Argument

Problem

You would like to write a single decorator that can be used without arguments, such as `@decorator`, or with optional arguments, such as `@decorator(x,y,z)`. However, there seems to be no straightforward way to do it due to differences in calling conventions between simple decorators and decorators taking arguments.

Solution

Here is a variant of the logging code shown in "Defining a Decorator with User Adjustable Attributes" that defines such a decorator:

```
from functools import wraps, partial
import logging

def logged(func=None, *, level=logging.DEBUG, name=None, message=None):
    if func is None:
        return partial(logged, level=level, name=name, message=message)

    logname = name if name else func.__module__
    log = logging.getLogger(logname)
    logmsg = message if message else func.__name__
    @wraps(func)
```

```
    def wrapper(*args, **kwargs):
        log.log(level, logmsg)
        return func(*args, **kwargs)
    return wrapper

# Example use
@logged
def add(x, y):
    return x + y

@logged(level=logging.CRITICAL, name='example')
def spam():
    print('Spam!')
```

As you can see from the example, the decorator can be used in both a simple form (i.e., `@logged`) or with optional arguments supplied (i.e., `@logged(level=logging.CRITICAL, name='example')`).

Discussion

The problem addressed by this recipe is really one of programming consistency. When using decorators, most programmers are used to applying them without any arguments at all or with arguments, as shown in the example. Technically speaking, a decorator where all arguments are optional could be applied, like this:

```
@logged()
def add(x, y):
    return x+y
```

However, this is not a form that's especially common, and might lead to common usage errors if programmers forget to add the extra parentheses. The recipe simply makes the decorator work with or without parentheses in a consistent way.

To understand how the code works, you need to have a firm understanding of how decorators get applied to functions and their calling conventions. For a simple decorator such as this:

```
# Example use
@logged
def add(x, y):
    return x + y
```

The calling sequence is as follows:

```
def add(x, y):
    return x + y
add = logged(add)
```

In this case, the function to be wrapped is simply passed to `logged` as the first argument. Thus, in the solution, the first argument of `logged()` is the function being wrapped. All of the other arguments must have default values.

For a decorator taking arguments such as this:

```
@logged(level=logging.CRITICAL, name='example')
def spam():
    print('Spam!')
```

The calling sequence is as follows:

```
def spam():
    print('Spam!')
spam = logged(level=logging.CRITICAL, name='example')(spam)
```

On the initial invocation of `logged()`, the function to be wrapped is not passed. Thus, in the decorator, it has to be optional. This, in turn, forces the other arguments to be specified by keyword. Furthermore, when arguments are passed, a decorator is supposed to return a function that accepts the function and wraps it (see "Defining a Decorator with User Adjustable Attributes"). To do this, the solution uses a clever trick involving `functools.partial`. Specifically, it simply returns a partially applied version of itself where all arguments are fixed except for the function to be wrapped. See "Making an N-Argument Callable Work As a Callable with Fewer Arguments" for more details about using `partial()`.

Enforcing Type Checking on a Function Using a Decorator

Problem

You want to optionally enforce type checking of function arguments as a kind of assertion or contract.

Solution

Before showing the solution code, the aim of this recipe is to have a means of enforcing type contracts on the input arguments to a function. Here is a short example that illustrates the idea:

```
>>> @typeassert(int, int)
... def add(x, y):
...     return x + y
...
>>>
>>> add(2, 3)
5
>>> add(2, 'hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "contract.py", line 33, in wrapper
TypeError: Argument y must be <class 'int'>
>>>
```

Now, here is an implementation of the `@typeassert` decorator:

```
from inspect import signature
from functools import wraps

def typeassert(*ty_args, **ty_kwargs):
    def decorate(func):
        # If in optimized mode, disable type checking
        if not __debug__:
            return func

        # Map function argument names to supplied types
        sig = signature(func)
        bound_types = sig.bind_partial(*ty_args, **ty_kwargs).arguments

        @wraps(func)
```

```
        def wrapper(*args, **kwargs):
            bound_values = sig.bind(*args, **kwargs)
            # Enforce type assertions across supplied arguments
            for name, value in bound_values.arguments.items():
                if name in bound_types:
                    if not isinstance(value, bound_types[name]):
                      raise TypeError(
                        'Argument {} must be {}'.format(name, bound_types[name])
                        )
            return func(*args, **kwargs)
        return wrapper
    return decorate
```

You will find that this decorator is rather flexible, allowing types to be specified for all or a subset of a function's arguments. Moreover, types can be specified by position or by keyword. Here is an example:

```
>>> @typeassert(int, z=int)
... def spam(x, y, z=42):
...     print(x, y, z)
...
>>> spam(1, 2, 3)
1 2 3
>>> spam(1, 'hello', 3)
1 hello 3
>>> spam(1, 'hello', 'world')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "contract.py", line 33, in wrapper
TypeError: Argument z must be <class 'int'>
>>>
```

Discussion

This recipe is an advanced decorator example that introduces a number of important and useful concepts.

First, one aspect of decorators is that they only get applied once, at the time of function definition. In certain cases, you may want to disable the functionality added by a decorator. To do this, simply have your decorator function return the function unwrapped. In the solution, the following code fragment returns the function unmodified if the value of the global `__debug__` variable is set to `False` (as is the case when Python executes in optimized mode with the `-O` or `-OO` options to the interpreter):

```
...
def decorate(func):
    # If in optimized mode, disable type checking
    if not __debug__:
        return func
    ...
```

Next, a tricky part of writing this decorator is that it involves examining and working with the argument signature of the function being wrapped. Your tool of choice here should be the `inspect.signature()` function. Simply stated, it allows you to extract signature information from a callable. For example:

```
>>> from inspect import signature
>>> def spam(x, y, z=42):
...     pass
...
>>> sig = signature(spam)
```

```
>>> print(sig)
(x, y, z=42)
>>> sig.parameters
mappingproxy(OrderedDict([('x', <Parameter at 0x10077a050 'x'>),
('y', <Parameter at 0x10077a158 'y'>), ('z', <Parameter at 0x10077a1b0 'z'>)]))
>>> sig.parameters['z'].name
'z'
>>> sig.parameters['z'].default
42
>>> sig.parameters['z'].kind
<_ParameterKind: 'POSITIONAL_OR_KEYWORD'>
>>>
```

In the first part of our decorator, we use the `bind_partial()` method of signatures to perform a partial binding of the supplied types to argument names. Here is an example of what happens:

```
>>> bound_types = sig.bind_partial(int,z=int)
>>> bound_types
<inspect.BoundArguments object at 0x10069bb50>
>>> bound_types.arguments
OrderedDict([('x', <class 'int'>), ('z', <class 'int'>)])
>>>
```

In this partial binding, you will notice that missing arguments are simply ignored (i.e., there is no binding for argument `y`). However, the most important part of the binding is the creation of the ordered dictionary `bound_types.arguments`. This dictionary maps the argument names to the supplied values in the same order as the function signature. In the case of our decorator, this mapping contains the type assertions that we're going to enforce.

In the actual wrapper function made by the decorator, the `sig.bind()` method is used. `bind()` is like `bind_partial()` except that it does not allow for missing arguments. So, here is what happens:

```
>>> bound_values = sig.bind(1, 2, 3)
>>> bound_values.arguments
OrderedDict([('x', 1), ('y', 2), ('z', 3)])
>>>
```

Using this mapping, it is relatively easy to enforce the required assertions.

```
>>> for name, value in bound_values.arguments.items():
...     if name in bound_types.arguments:
...         if not isinstance(value, bound_types.arguments[name]):
...             raise TypeError()
...
>>>
```

A somewhat subtle aspect of the solution is that the assertions do not get applied to unsupplied arguments with default values. For example, this code works, even though the default value of `items` is of the "wrong" type:

```
>>> @typeassert(int, list)
... def bar(x, items=None):
...     if items is None:
...         items = []
...     items.append(x)
...     return items
>>> bar(2)
```

```
[2]
>>> bar(2,3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "contract.py", line 33, in wrapper
TypeError: Argument items must be <class 'list'>
>>> bar(4, [1, 2, 3])
[1, 2, 3, 4]
>>>
```

A final point of design discussion might be the use of decorator arguments versus function annotations. For example, why not write the decorator to look at annotations like this?

```
@typeassert
def spam(x:int, y, z:int = 42):
    print(x,y,z)
```

One possible reason for not using annotations is that each argument to a function can only have a single annotation assigned. Thus, if the annotations are used for type assertions, they can't really be used for anything else. Likewise, the `@typeassert` decorator won't work with functions that use annotations for a different purpose. By using decorator arguments, as shown in the solution, the decorator becomes a lot more general purpose and can be used with any function whatsoever—even functions that use annotations.

More information about function signature objects can be found in PEP 362, as well as the documentation for the `inspect` module. "Enforcing an Argument Signature on *args and **kwargs" also has an additional example.

Defining Decorators As Part of a Class

Problem

You want to define a decorator inside a class definition and apply it to other functions or methods.

Solution

Defining a decorator inside a class is straightforward, but you first need to sort out the manner in which the decorator will be applied. Specifically, whether it is applied as an instance or a class method. Here is an example that illustrates the difference:

```
from functools import wraps

class A:
    # Decorator as an instance method
    def decorator1(self, func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            print('Decorator 1')
            return func(*args, **kwargs)
        return wrapper

    # Decorator as a class method
    @classmethod
    def decorator2(cls, func):
```

```
        @wraps(func)
        def wrapper(*args, **kwargs):
            print('Decorator 2')
            return func(*args, **kwargs)
        return wrapper
```

Here is an example of how the two decorators would be applied:

```
# As an instance method
a = A()

@a.decorator1
def spam():
    pass

# As a class method
@A.decorator2
def grok():
    pass
```

If you look carefully, you'll notice that one is applied from an instance `a` and the other is applied from the class `A`.

Discussion

Defining decorators in a class might look odd at first glance, but there are examples of this in the standard library. In particular, the built-in `@property` decorator is actually a class with `getter()`, `setter()`, and `deleter()` methods that each act as a decorator. For example:

```
class Person:
    # Create a property instance
    first_name = property()

    # Apply decorator methods
    @first_name.getter
    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
```

The key reason why it's defined in this way is that the various decorator methods are manipulating state on the associated `property` instance. So, if you ever had a problem where decorators needed to record or combine information behind the scenes, it's a sensible approach.

A common confusion when writing decorators in classes is getting tripped up by the proper use of the extra `self` or `cls` arguments in the decorator code itself. Although the outermost decorator function, such as `decorator1()` or `decorator2()`, needs to provide a `self` or `cls` argument (since they're part of a class), the wrapper function created inside doesn't generally need to include an extra argument. This is why the `wrapper()` function created in both decorators doesn't include a `self` argument. The only time you would ever need this argument is in situations where you actually needed to access parts of an instance in the wrapper. Otherwise, you just don't have to worry about it.

A final subtle facet of having decorators defined in a class concerns their potential use with inheritance. For example, suppose you want to apply one of the decorators defined in class `A` to methods defined in a subclass `B`. To do that, you would need to write code like this:

```
class B(A):
    @A.decorator2
    def bar(self):
        pass
```

In particular, the decorator in question has to be defined as a class method and you have to explicitly use the name of the superclass `A` when applying it. You can't use a name such as `@B.decorator2`, because at the time of method definition, class `B` has not yet been created.

Defining Decorators As Classes

Problem

You want to wrap functions with a decorator, but the result is going to be a callable instance. You need your decorator to work both inside and outside class definitions.

Solution

To define a decorator as an instance, you need to make sure it implements the `__call__()` and `__get__()` methods. For example, this code defines a class that puts a simple profiling layer around another function:

```
import types
from functools import wraps

class Profiled:
    def __init__(self, func):
        wraps(func)(self)
        self.ncalls = 0

    def __call__(self, *args, **kwargs):
        self.ncalls += 1
        return self.__wrapped__(*args, **kwargs)

    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            return types.MethodType(self, instance)
```

To use this class, you use it like a normal decorator, either inside or outside of a class:

```
@Profiled
def add(x, y):
    return x + y

class Spam:
    @Profiled
    def bar(self, x):
        print(self, x)
```

Here is an interactive session that shows how these functions work:

```
>>> add(2, 3)
5
>>> add(4, 5)
9
>>> add.ncalls
2
>>> s = Spam()
>>> s.bar(1)
<__main__.Spam object at 0x10069e9d0> 1
>>> s.bar(2)
<__main__.Spam object at 0x10069e9d0> 2
>>> s.bar(3)
<__main__.Spam object at 0x10069e9d0> 3
>>> Spam.bar.ncalls
3
```

Discussion

Defining a decorator as a class is usually straightforward. However, there are some rather subtle details that deserve more explanation, especially if you plan to apply the decorator to instance methods.

First, the use of the `functools.wraps()` function serves the same purpose here as it does in normal decorators—namely to copy important metadata from the wrapped function to the callable instance.

Second, it is common to overlook the `__get__()` method shown in the solution. If you omit the `__get__()` and keep all of the other code the same, you'll find that bizarre things happen when you try to invoke decorated instance methods. For example:

```
>>> s = Spam()
>>> s.bar(3)
Traceback (most recent call last):
...
TypeError: spam() missing 1 required positional argument: 'x'
```

The reason it breaks is that whenever functions implementing methods are looked up in a class, their `__get__()` method is invoked as part of the descriptor protocol, which is described in "Creating a New Kind of Class or Instance Attribute". In this case, the purpose of `__get__()` is to create a bound method object (which ultimately supplies the `self` argument to the method). Here is an example that illustrates the underlying mechanics:

```
>>> s = Spam()
>>> def grok(self, x):
...     pass
...
>>> grok.__get__(s, Spam)
<bound method Spam.grok of <__main__.Spam object at 0x100671e90>>
>>>
```

In this recipe, the `__get__()` method is there to make sure bound method objects get created properly. `type.MethodType()` creates a bound method manually for use here. Bound methods only get created if an instance is being used. If the method is accessed on a class, the `instance` argument to `__get__()` is set to `None` and the `Profiled` instance itself is just returned. This makes it possible for someone to extract its `ncalls` attribute, as shown.

If you want to avoid some of this of this mess, you might consider an alternative formulation of the

decorator using closures and `nonlocal` variables, as described in "Defining a Decorator with User Adjustable Attributes". For example:

```
import types
from functools import wraps

def profiled(func):
    ncalls = 0
    @wraps(func)
    def wrapper(*args, **kwargs):
        nonlocal ncalls
        ncalls += 1
        return func(*args, **kwargs)
    wrapper.ncalls = lambda: ncalls
    return wrapper

# Example
@profiled
def add(x, y):
    return x + y
```

This example almost works in exactly the same way except that access to `ncalls` is now provided through a function attached as a function attribute. For example:

```
>>> add(2, 3)
5
>>> add(4, 5)
9
>>> add.ncalls()
2
>>>
```

Applying Decorators to Class and Static Methods

Problem

You want to apply a decorator to a class or static method.

Solution

Applying decorators to class and static methods is straightforward, but make sure that your decorators are applied before `@classmethod` or `@staticmethod`. For example:

```
import time
from functools import wraps

# A simple decorator
def timethis(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        r = func(*args, **kwargs)
        end = time.time()
        print(end-start)
        return r
    return wrapper
```

```
# Class illustrating application of the decorator to different kinds of methods
class Spam:
    @timethis
    def instance_method(self, n):
        print(self, n)
        while n > 0:
            n -= 1

    @classmethod
    @timethis
    def class_method(cls, n):
        print(cls, n)
        while n > 0:
            n -= 1

    @staticmethod
    @timethis
    def static_method(n):
        print(n)
        while n > 0:
            n -= 1
```

The resulting class and static methods should operate normally, but have the extra timing:

```
>>> s = Spam()
>>> s.instance_method(1000000)
<__main__.Spam object at 0x1006a6050> 1000000
0.11817407608032227
>>> Spam.class_method(1000000)
<class '__main__.Spam'> 1000000
0.11334395408630371
>>> Spam.static_method(1000000)
1000000
0.11740279197692871
>>>
```

Discussion

If you get the order of decorators wrong, you'll get an error. For example, if you use the following:

```
class Spam:
    ...
    @timethis
    @staticmethod
    def static_method(n):
        print(n)
        while n > 0:
            n -= 1
```

Then the static method will crash:

```
>>> Spam.static_method(1000000)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "timethis.py", line 6, in wrapper
    start = time.time()
TypeError: 'staticmethod' object is not callable
>>>
```

The problem here is that `@classmethod` and `@staticmethod` don't actually create objects that are directly callable. Instead, they create special descriptor objects, as described in "Creating a New Kind of Class or Instance Attribute". Thus, if you try to use them like functions in another decorator, the decorator will crash. Making sure that these decorators appear first in the decorator list fixes the problem.

One situation where this recipe is of critical importance is in defining class and static methods in abstract base classes, as described in "Defining an Interface or Abstract Base Class". For example, if you want to define an abstract class method, you can use this code:

```
from abc import ABCMeta, abstractmethod

class A(metaclass=ABCMeta):
    @classmethod
    @abstractmethod
    def method(cls):
        pass
```

In this code, the order of `@classmethod` and `@abstractmethod` matters. If you flip the two decorators around, everything breaks.

Writing Decorators That Add Arguments to Wrapped Functions

Problem

You want to write a decorator that adds an extra argument to the calling signature of the wrapped function. However, the added argument can't interfere with the existing calling conventions of the function.

Solution

Extra arguments can be injected into the calling signature using keyword-only arguments. Consider the following decorator:

```
from functools import wraps

def optional_debug(func):
    @wraps(func)
    def wrapper(*args, debug=False, **kwargs):
        if debug:
            print('Calling', func.__name__)
        return func(*args, **kwargs)
    return wrapper
```

Here is an example of how the decorator works:

```
>>> @optional_debug
... def spam(a,b,c):
...     print(a,b,c)
...
>>> spam(1,2,3)
1 2 3
>>> spam(1,2,3, debug=True)
Calling spam
1 2 3
>>>
```

Discussion

Adding arguments to the signature of wrapped functions is not the most common example of using decorators. However, it might be a useful technique in avoiding certain kinds of code replication patterns. For example, if you have code like this:

```
def a(x, debug=False):
    if debug:
        print('Calling a')
    ...

def b(x, y, z, debug=False):
    if debug:
        print('Calling b')
    ...

def c(x, y, debug=False):
    if debug:
        print('Calling c')
    ...
```

You can refactor it into the following:

```
@optional_debug
def a(x):
    ...

@optional_debug
def b(x, y, z):
    ...

@optional_debug
def c(x, y):
    ...
```

The implementation of this recipe relies on the fact that keyword-only arguments are easy to add to functions that also accept `*args` and `**kwargs` parameters. By using a keyword-only argument, it gets singled out as a special case and removed from subsequent calls that only use the remaining positional and keyword arguments.

One tricky part here concerns a potential name clash between the added argument and the arguments of the function being wrapped. For example, if the `@optional_debug` decorator was applied to a function that already had a `debug` argument, then it would break. If that's a concern, an extra check could be added:

```
from functools import wraps
import inspect

def optional_debug(func):
    if 'debug' in inspect.getargspec(func).args:
        raise TypeError('debug argument already defined')

    @wraps(func)
    def wrapper(*args, debug=False, **kwargs):
        if debug:
            print('Calling', func.__name__)
        return func(*args, **kwargs)
    return wrapper
```

A final refinement to this recipe concerns the proper management of function signatures. An astute programmer will realize that the signature of wrapped functions is wrong. For example:

```
>>> @optional_debug
... def add(x,y):
...     return x+y
...
>>> import inspect
>>> print(inspect.signature(add))
(x, y)
>>>
```

This can be fixed by making the following modification:

```
from functools import wraps
import inspect

def optional_debug(func):
    if 'debug' in inspect.getargspec(func).args:
        raise TypeError('debug argument already defined')

    @wraps(func)
    def wrapper(*args, debug=False, **kwargs):
        if debug:
            print('Calling', func.__name__)
        return func(*args, **kwargs)

    sig = inspect.signature(func)
    parms = list(sig.parameters.values())
    parms.append(inspect.Parameter('debug',
                                   inspect.Parameter.KEYWORD_ONLY,
                                   default=False))
    wrapper.__signature__ = sig.replace(parameters=parms)
    return wrapper
```

With this change, the signature of the wrapper will now correctly reflect the presence of the `debug` argument. For example:

```
>>> @optional_debug
... def add(x,y):
...     return x+y
...
>>> print(inspect.signature(add))
(x, y, *, debug=False)
>>> add(2,3)
5
>>>
```

See "Enforcing an Argument Signature on *args and **kwargs" for more information about function signatures.

Using Decorators to Patch Class Definitions

Problem

You want to inspect or rewrite portions of a class definition to alter its behavior, but without using

inheritance or metaclasses.

Solution

This might be a perfect use for a class decorator. For example, here is a class decorator that rewrites the `__getattribute__` special method to perform logging.

```
def log_getattribute(cls):
    # Get the original implementation
    orig_getattribute = cls.__getattribute__

    # Make a new definition
    def new_getattribute(self, name):
        print('getting:', name)
        return orig_getattribute(self, name)

    # Attach to the class and return
    cls.__getattribute__ = new_getattribute
    return cls

# Example use
@log_getattribute
class A:
    def __init__(self,x):
        self.x = x
    def spam(self):
        pass
```

Here is what happens if you try to use the class in the solution:

```
>>> a = A(42)
>>> a.x
getting: x
42
>>> a.spam()
getting: spam
>>>
```

Discussion

Class decorators can often be used as a straightforward alternative to other more advanced techniques involving mixins or metaclasses. For example, an alternative implementation of the solution might involve inheritance, as in the following:

```
class LoggedGetattribute:
    def __getattribute__(self, name):
        print('getting:', name)
        return super().__getattribute__(name)

# Example:
class A(LoggedGetattribute):
    def __init__(self,x):
        self.x = x
    def spam(self):
        pass
```

This works, but to understand it, you have to have some awareness of the method resolution order,

`super()`, and other aspects of inheritance, as described in "Calling a Method on a Parent Class". In some sense, the class decorator solution is much more direct in how it operates, and it doesn't introduce new dependencies into the inheritance hierarchy. As it turns out, it's also just a bit faster, due to not relying on the `super()` function.

If you are applying multiple class decorators to a class, the application order might matter. For example, a decorator that replaces a method with an entirely new implementation would probably need to be applied before a decorator that simply wraps an existing method with some extra logic.

See "Implementing a Data Model or Type System" for another example of class decorators in action.

Using a Metaclass to Control Instance Creation

Problem

You want to change the way in which instances are created in order to implement singletons, caching, or other similar features.

Solution

As Python programmers know, if you define a class, you call it like a function to create instances. For example:

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

a = Spam('Guido')
b = Spam('Diana')
```

If you want to customize this step, you can do it by defining a metaclass and reimplementing its `__call__()` method in some way. To illustrate, suppose that you didn't want anyone creating instances at all:

```
class NoInstances(type):
    def __call__(self, *args, **kwargs):
        raise TypeError("Can't instantiate directly")

# Example
class Spam(metaclass=NoInstances):
    @staticmethod
    def grok(x):
        print('Spam.grok')
```

In this case, users can call the defined static method, but it's impossible to create an instance in the normal way. For example:

```
>>> Spam.grok(42)
Spam.grok
>>> s = Spam()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "example1.py", line 7, in __call__
    raise TypeError("Can't instantiate directly")
```

```
TypeError: Can't instantiate directly
>>>
```

Now, suppose you want to implement the singleton pattern (i.e., a class where only one instance is ever created). That is also relatively straightforward, as shown here:

```
class Singleton(type):
    def __init__(self, *args, **kwargs):
        self.__instance = None
        super().__init__(*args, **kwargs)

    def __call__(self, *args, **kwargs):
        if self.__instance is None:
            self.__instance = super().__call__(*args, **kwargs)
            return self.__instance
        else:
            return self.__instance

# Example
class Spam(metaclass=Singleton):
    def __init__(self):
        print('Creating Spam')
```

In this case, only one instance ever gets created. For example:

```
>>> a = Spam()
Creating Spam
>>> b = Spam()
>>> a is b
True
>>> c = Spam()
>>> a is c
True
>>>
```

Finally, suppose you want to create cached instances, as described in "Creating Cached Instances". Here's a metaclass that implements it:

```
import weakref

class Cached(type):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.__cache = weakref.WeakValueDictionary()

    def __call__(self, *args):
        if args in self.__cache:
            return self.__cache[args]
        else:
            obj = super().__call__(*args)
            self.__cache[args] = obj
            return obj

# Example
class Spam(metaclass=Cached):
    def __init__(self, name):
        print('Creating Spam({!r})'.format(name))
        self.name = name
```

Here's an example showing the behavior of this class:

```
>>> a = Spam('Guido')
Creating Spam('Guido')
>>> b = Spam('Diana')
Creating Spam('Diana')
>>> c = Spam('Guido')       # Cached
>>> a is b
False
>>> a is c                  # Cached value returned
True
>>>
```

Discussion

Using a metaclass to implement various instance creation patterns can often be a much more elegant approach than other solutions not involving metaclasses. For example, if you didn't use a metaclass, you might have to hide the classes behind some kind of extra factory function. For example, to get a singleton, you might use a hack such as the following:

```
class _Spam:
    def __init__(self):
        print('Creating Spam')

_spam_instance = None
def Spam():
    global _spam_instance
    if _spam_instance is not None:
        return _spam_instance
    else:
        _spam_instance = _Spam()
        return _spam_instance
```

Although the solution involving metaclasses involves a much more advanced concept, the resulting code feels cleaner and less hacked together.

See "Creating Cached Instances" for more information on creating cached instances, weak references, and other details.

Capturing Class Attribute Definition Order

Problem

You want to automatically record the order in which attributes and methods are defined inside a class body so that you can use it in various operations (e.g., serializing, mapping to databases, etc.).

Solution

Capturing information about the body of class definition is easily accomplished through the use of a metaclass. Here is an example of a metaclass that uses an `OrderedDict` to capture definition order of descriptors:

```
from collections import OrderedDict
```

```
# A set of descriptors for various types
class Typed:
    _expected_type = type(None)
    def __init__(self, name=None):
        self._name = 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

class Integer(Typed):
    _expected_type = int

class Float(Typed):
    _expected_type = float

class String(Typed):
    _expected_type = str

# Metaclass that uses an OrderedDict for class body
class OrderedMeta(type):
    def __new__(cls, clsname, bases, clsdict):
        d = dict(clsdict)
        order = []
        for name, value in clsdict.items():
            if isinstance(value, Typed):
                value._name = name
                order.append(name)
        d['_order'] = order
        return type.__new__(cls, clsname, bases, d)

    @classmethod
    def __prepare__(cls, clsname, bases):
        return OrderedDict()
```

In this metaclass, the definition order of descriptors is captured by using an `OrderedDict` during the execution of the class body. The resulting order of names is then extracted from the dictionary and stored into a class attribute `_order`. This can then be used by methods of the class in various ways. For example, here is a simple class that uses the ordering to implement a method for serializing the instance data as a line of CSV data:

```
class Structure(metaclass=OrderedMeta):
    def as_csv(self):
        return ','.join(str(getattr(self,name)) for name in self._order)

# Example use
class Stock(Structure):
    name = String()
    shares = Integer()
    price = Float()
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price
```

Here is an interactive session illustrating the use of the `Stock` class in the example:

```
>>> s = Stock('GOOG',100,490.1)
>>> s.name
'GOOG'
>>> s.as_csv()
'GOOG,100,490.1'
>>> t = Stock('AAPL','a lot', 610.23)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "dupmethod.py", line 34, in __init__
TypeError: shares expects <class 'int'>
>>>
```

Discussion

The entire key to this recipe is the `__prepare__()` method, which is defined in the `OrderedMeta` metaclass. This method is invoked immediately at the start of a class definition with the class name and base classes. It must then return a mapping object to use when processing the class body. By returning an `OrderedDict` instead of a normal dictionary, the resulting definition order is easily captured.

It is possible to extend this functionality even further if you are willing to make your own dictionary-like objects. For example, consider this variant of the solution that rejects duplicate definitions:

```
from collections import OrderedDict

class NoDupOrderedDict(OrderedDict):
    def __init__(self, clsname):
        self.clsname = clsname
        super().__init__()
    def __setitem__(self, name, value):
        if name in self:
            raise TypeError('{} already defined in {}'.format(name, self.clsname))
        super().__setitem__(name, value)

class OrderedMeta(type):
    def __new__(cls, clsname, bases, clsdict):
        d = dict(clsdict)
        d['_order'] = [name for name in clsdict if name[0] != '_']
        return type.__new__(cls, clsname, bases, d)

    @classmethod
    def __prepare__(cls, clsname, bases):
        return NoDupOrderedDict(clsname)
```

Here's what happens if you use this metaclass and make a class with duplicate entries:

```
>>> class A(metaclass=OrderedMeta):
...     def spam(self):
...             pass
...     def spam(self):
...             pass
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in A
  File "dupmethod2.py", line 25, in __setitem__
    (name, self.clsname))
TypeError: spam already defined in A
>>>
```

A final important part of this recipe concerns the treatment of the modified dictionary in the metaclass `__new__()` method. Even though the class was defined using an alternative dictionary, you still have to convert this dictionary to a proper `dict` instance when making the final class object. This is the purpose of the `d = dict(clsdict)` statement.

Being able to capture definition order is a subtle but important feature for certain kinds of applications. For instance, in an object relational mapper, classes might be written in a manner similar to that shown in the example:

```
class Stock(Model):
    name = String()
    shares = Integer()
    price = Float()
```

Underneath the covers, the code might want to capture the definition order to map objects to tuples or rows in a database table (e.g., similar to the functionality of the `as_csv()` method in the example). The solution shown is very straightforward and often simpler than alternative approaches (which typically involve maintaining hidden counters within the descriptor classes).

Defining a Metaclass That Takes Optional Arguments

Problem

You want to define a metaclass that allows class definitions to supply optional arguments, possibly to control or configure aspects of processing during type creation.

Solution

When defining classes, Python allows a metaclass to be specified using the `metaclass` keyword argument in the `class` statement. For example, with abstract base classes:

```
from abc import ABCMeta, abstractmethod

class IStream(metaclass=ABCMeta):
    @abstractmethod
    def read(self, maxsize=None):
        pass

    @abstractmethod
    def write(self, data):
        pass
```

However, in custom metaclasses, additional keyword arguments can be supplied, like this:

```
class Spam(metaclass=MyMeta, debug=True, synchronize=True):
    ...
```

To support such keyword arguments in a metaclass, make sure you define them on the `__prepare__()`, `__new__()`, and `__init__()` methods using keyword-only arguments, like this:

```
class MyMeta(type):
    # Optional
    @classmethod
```

```
    def __prepare__(cls, name, bases, *, debug=False, synchronize=False):
        # Custom processing
        ...
        return super().__prepare__(name, bases)

    # Required
    def __new__(cls, name, bases, ns, *, debug=False, synchronize=False):
        # Custom processing
        ...
        return super().__new__(cls, name, bases, ns)

    # Required
    def __init__(self, name, bases, ns, *, debug=False, synchronize=False):
        # Custom processing
        ...
        super().__init__(name, bases, ns)
```

Discussion

Adding optional keyword arguments to a metaclass requires that you understand all of the steps involved in class creation, because the extra arguments are passed to every method involved. The `__prepare__()` method is called first and used to create the class namespace prior to the body of any class definition being processed. Normally, this method simply returns a dictionary or other mapping object. The `__new__()` method is used to instantiate the resulting type object. It is called after the class body has been fully executed. The `__init__()` method is called last and used to perform any additional initialization steps.

When writing metaclasses, it is somewhat common to only define a `__new__()` or `__init__()` method, but not both. However, if extra keyword arguments are going to be accepted, then both methods must be provided and given compatible signatures. The default `__prepare__()` method accepts any set of keyword arguments, but ignores them. You only need to define it yourself if the extra arguments would somehow affect management of the class namespace creation.

The use of keyword-only arguments in this recipe reflects the fact that such arguments will only be supplied by keyword during class creation.

The specification of keyword arguments to configure a metaclass might be viewed as an alternative to using class variables for a similar purpose. For example:

```
class Spam(metaclass=MyMeta):
    debug = True
    synchronize = True
    ...
```

The advantage to supplying such parameters as an argument is that they don’t pollute the class namespace with extra names that only pertain to class creation and not the subsequent execution of statements in the class. In addition, they are available to the `__prepare__()` method, which runs prior to processing any statements in the class body. Class variables, on the other hand, would only be accessible in the `__new__()` and `__init__()` methods of a metaclass.

Enforcing an Argument Signature on *args and **kwargs

Problem

You've written a function or method that uses `*args` and `**kwargs`, so that it can be general purpose, but you would also like to check the passed arguments to see if they match a specific function calling signature.

Solution

For any problem where you want to manipulate function calling signatures, you should use the signature features found in the `inspect` module. Two classes, `Signature` and `Parameter`, are of particular interest here. Here is an interactive example of creating a function signature:

```
>>> from inspect import Signature, Parameter
>>> # Make a signature for a func(x, y=42, *, z=None)
>>> parms = [ Parameter('x', Parameter.POSITIONAL_OR_KEYWORD),
...           Parameter('y', Parameter.POSITIONAL_OR_KEYWORD, default=42),
...           Parameter('z', Parameter.KEYWORD_ONLY, default=None) ]
>>> sig = Signature(parms)
>>> print(sig)
(x, y=42, *, z=None)
>>>
```

Once you have a signature object, you can easily bind it to `*args` and `**kwargs` using the signature's `bind()` method, as shown in this simple example:

```
>>> def func(*args, **kwargs):
...     bound_values = sig.bind(*args, **kwargs)
...     for name, value in bound_values.arguments.items():
...         print(name,value)
...
>>> # Try various examples
>>> func(1, 2, z=3)
x 1
y 2
z 3
>>> func(1)
x 1
>>> func(1, z=3)
x 1
z 3
>>> func(y=2, x=1)
x 1
y 2
>>> func(1, 2, 3, 4)
Traceback (most recent call last):
...
  File "/usr/local/lib/python3.3/inspect.py", line 1972, in _bind
    raise TypeError('too many positional arguments')
TypeError: too many positional arguments
>>> func(y=2)
Traceback (most recent call last):
...
  File "/usr/local/lib/python3.3/inspect.py", line 1961, in _bind
    raise TypeError(msg) from None
TypeError: 'x' parameter lacking default value
>>> func(1, y=2, x=3)
Traceback (most recent call last):
...
  File "/usr/local/lib/python3.3/inspect.py", line 1985, in _bind
    '{arg!r}'.format(arg=param.name))
```

```
TypeError: multiple values for argument 'x'
>>>
```

As you can see, the binding of a signature to the passed arguments enforces all of the usual function calling rules concerning required arguments, defaults, duplicates, and so forth.

Here is a more concrete example of enforcing function signatures. In this code, a base class has defined an extremely general-purpose version of `__init__()`, but subclasses are expected to supply an expected signature.

```
from inspect import Signature, Parameter

def make_sig(*names):
    parms = [Parameter(name, Parameter.POSITIONAL_OR_KEYWORD)
             for name in names]
    return Signature(parms)

class Structure:
    __signature__ = make_sig()
    def __init__(self, *args, **kwargs):
        bound_values = self.__signature__.bind(*args, **kwargs)
        for name, value in bound_values.arguments.items():
            setattr(self, name, value)

# Example use
class Stock(Structure):
    __signature__ = make_sig('name', 'shares', 'price')

class Point(Structure):
    __signature__ = make_sig('x', 'y')
```

Here is an example of how the `Stock` class works:

```
>>> import inspect
>>> print(inspect.signature(Stock))
(name, shares, price)
>>> s1 = Stock('ACME', 100, 490.1)
>>> s2 = Stock('ACME', 100)
Traceback (most recent call last):
...
TypeError: 'price' parameter lacking default value
>>> s3 = Stock('ACME', 100, 490.1, shares=50)
Traceback (most recent call last):
...
TypeError: multiple values for argument 'shares'
>>>
```

Discussion

The use of functions involving `*args` and `**kwargs` is very common when trying to make general-purpose libraries, write decorators or implement proxies. However, one downside of such functions is that if you want to implement your own argument checking, it can quickly become an unwieldy mess. As an example, see "Simplifying the Initialization of Data Structures". The use of a signature object simplifies this.

In the last example of the solution, it might make sense to create signature objects through the use of a custom metaclass. Here is an alternative implementation that shows how to do this:

```
from inspect import Signature, Parameter

def make_sig(*names):
    parms = [Parameter(name, Parameter.POSITIONAL_OR_KEYWORD)
             for name in names]
    return Signature(parms)

class StructureMeta(type):
    def __new__(cls, clsname, bases, clsdict):
        clsdict['__signature__'] = make_sig(*clsdict.get('_fields',[]))
        return super().__new__(cls, clsname, bases, clsdict)

class Structure(metaclass=StructureMeta):
    _fields = []
    def __init__(self, *args, **kwargs):
        bound_values = self.__signature__.bind(*args, **kwargs)
        for name, value in bound_values.arguments.items():
            setattr(self, name, value)

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

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

When defining custom signatures, it is often useful to store the signature in a special attribute `__signature__`, as shown. If you do this, code that uses the `inspect` module to perform introspection will see the signature and report it as the calling convention. For example:

```
>>> import inspect
>>> print(inspect.signature(Stock))
(name, shares, price)
>>> print(inspect.signature(Point))
(x, y)
>>>
```

Enforcing Coding Conventions in Classes

Problem

Your program consists of a large class hierarchy and you would like to enforce certain kinds of coding conventions (or perform diagnostics) to help maintain programmer sanity.

Solution

If you want to monitor the definition of classes, you can often do it by defining a metaclass. A basic metaclass is usually defined by inheriting from `type` and redefining its `__new__()` method or `__init__()` method. For example:

```
class MyMeta(type):
    def __new__(self, clsname, bases, clsdict):
        # clsname is name of class being defined
        # bases is tuple of base classes
        # clsdict is class dictionary
        return super().__new__(cls, clsname, bases, clsdict)
```

Alternatively, if `__init__()` is defined:

```
class MyMeta(type):
    def __init__(self, clsname, bases, clsdict):
        super().__init__(clsname, bases, clsdict)
        # clsname is name of class being defined
        # bases is tuple of base classes
        # clsdict is class dictionary
```

To use a metaclass, you would generally incorporate it into a top-level base class from which other objects inherit. For example:

```
class Root(metaclass=MyMeta):
    pass

class A(Root):
    pass

class B(Root):
    pass
```

A key feature of a metaclass is that it allows you to examine the contents of a class at the time of definition. Inside the redefined `__init__()` method, you are free to inspect the class dictionary, base classes, and more. Moreover, once a metaclass has been specified for a class, it gets inherited by all of the subclasses. Thus, a sneaky framework builder can specify a metaclass for one of the top-level classes in a large hierarchy and capture the definition of all classes under it.

As a concrete albeit whimsical example, here is a metaclass that rejects any class definition containing methods with mixed-case names (perhaps as a means for annoying Java programmers):

```
class NoMixedCaseMeta(type):
    def __new__(cls, clsname, bases, clsdict):
        for name in clsdict:
            if name.lower() != name:
                raise TypeError('Bad attribute name: ' + name)
        return super().__new__(cls, clsname, bases, clsdict)

class Root(metaclass=NoMixedCaseMeta):
    pass

class A(Root):
    def foo_bar(self):      # Ok
        pass

class B(Root):
    def fooBar(self):       # TypeError
        pass
```

As a more advanced and useful example, here is a metaclass that checks the definition of redefined methods to make sure they have the same calling signature as the original method in the superclass.

```
from inspect import signature
import logging

class MatchSignaturesMeta(type):
    def __init__(self, clsname, bases, clsdict):
        super().__init__(clsname, bases, clsdict)
        sup = super(self, self)
```

```
        for name, value in clsdict.items():
            if name.startswith('_') or not callable(value):
                continue
            # Get the previous definition (if any) and compare the signatures
            prev_dfn = getattr(sup,name,None)
            if prev_dfn:
                prev_sig = signature(prev_dfn)
                val_sig = signature(value)
                if prev_sig != val_sig:
                    logging.warning('Signature mismatch in %s. %s != %s',
                                value.__qualname__, prev_sig, val_sig)

# Example
class Root(metaclass=MatchSignaturesMeta):
    pass

class A(Root):
    def foo(self, x, y):
        pass

    def spam(self, x, *, z):
        pass

# Class with redefined methods, but slightly different signatures
class B(A):
    def foo(self, a, b):
        pass

    def spam(self,x,z):
        pass
```

If you run this code, you will get output such as the following:

```
    WARNING:root:Signature mismatch in B.spam. (self, x, *, z) != (self, x, z)
    WARNING:root:Signature mismatch in B.foo. (self, x, y) != (self, a, b)
```

Such warnings might be useful in catching subtle program bugs. For example, code that relies on keyword argument passing to a method will break if a subclass changes the argument names.

Discussion

In large object-oriented programs, it can sometimes be useful to put class definitions under the control of a metaclass. The metaclass can observe class definitions and be used to alert programmers to potential problems that might go unnoticed (e.g., using slightly incompatible method signatures).

One might argue that such errors would be better caught by program analysis tools or IDEs. To be sure, such tools are useful. However, if you're creating a framework or library that's going to be used by others, you often don't have any control over the rigor of their development practices. Thus, for certain kinds of applications, it might make sense to put a bit of extra checking in a metaclass if such checking would result in a better user experience.

The choice of redefining `__new__()` or `__init__()` in a metaclass depends on how you want to work with the resulting class. `__new__()` is invoked prior to class creation and is typically used when a metaclass wants to alter the class definition in some way (by changing the contents of the class dictionary). The `__init__()` method is invoked after a class has been created, and is useful if you want to write code that works with the fully formed class object. In the last example, this is essential since it is using the `super()`

function to search for prior definitions. This only works once the class instance has been created and the underlying method resolution order has been set.

The last example also illustrates the use of Python's function signature objects. Essentially, the metaclass takes each callable definition in a class, searches for a prior definition (if any), and then simply compares their calling signatures using `inspect.signature()`.

Last, but not least, the line of code that uses `super(self, self)` is not a typo. When working with a metaclass, it's important to realize that the `self` is actually a class object. So, that statement is actually being used to find definitions located further up the class hierarchy that make up the parents of `self`.

Defining Classes Programmatically

Problem

You're writing code that ultimately needs to create a new class object. You've thought about emitting emit class source code to a string and using a function such as `exec()` to evaluate it, but you'd prefer a more elegant solution.

Solution

You can use the function `types.new_class()` to instantiate new class objects. All you need to do is provide the name of the class, tuple of parent classes, keyword arguments, and a callback that populates the class dictionary with members. For example:

```
# stock.py
# Example of making a class manually from parts

# Methods
def __init__(self, name, shares, price):
    self.name = name
    self.shares = shares
    self.price = price

def cost(self):
    return self.shares * self.price

cls_dict = {
    '__init__' : __init__,
    'cost' : cost,
}

# Make a class
import types

Stock = types.new_class('Stock', (), {}, lambda ns: ns.update(cls_dict))
Stock.__module__ = __name__
```

This makes a normal class object that works just like you expect:

```
>>> s = Stock('ACME', 50, 91.1)
>>> s
<stock.Stock object at 0x1006a9b10>
>>> s.cost()
```

```
4555.0
>>>
```

A subtle facet of the solution is the assignment to `Stock.__module__` after the call to `types.new_class()`. Whenever a class is defined, its `__module__` attribute contains the name of the module in which it was defined. This name is used to produce the output made by methods such as `__repr__()`. It's also used by various libraries, such as `pickle`. Thus, in order for the class you make to be "proper," you need to make sure this attribute is set accordingly.

If the class you want to create involves a different metaclass, it would be specified in the third argument to `types.new_class()`. For example:

```
>>> import abc
>>> Stock = types.new_class('Stock', (), {'metaclass': abc.ABCMeta},
...                         lambda ns: ns.update(cls_dict))
...
>>> Stock.__module__ = __name__
>>> Stock
<class '__main__.Stock'>
>>> type(Stock)
<class 'abc.ABCMeta'>
>>>
```

The third argument may also contain other keyword arguments. For example, a class definition like this

```
class Spam(Base, debug=True, typecheck=False):
    ...
```

would translate to a `new_class()` call similar to this:

```
Spam = types.new_class('Spam', (Base,),
                       {'debug': True, 'typecheck': False},
                       lambda ns: ns.update(cls_dict))
```

The fourth argument to `new_class()` is the most mysterious, but it is a function that receives the mapping object being used for the class namespace as input. This is normally a dictionary, but it's actually whatever object gets returned by the `__prepare__()` method, as described in "Capturing Class Attribute Definition Order". This function should add new entries to the namespace using the `update()` method (as shown) or other mapping operations.

Discussion

Being able to manufacture new class objects can be useful in certain contexts. One of the more familiar examples involves the `collections.namedtuple()` function. For example:

```
>>> Stock = collections.namedtuple('Stock', ['name', 'shares', 'price'])
>>> Stock
<class '__main__.Stock'>
>>>
```

`namedtuple()` uses `exec()` instead of the technique shown here. However, here is a simple variant that creates a class directly:

```
import operator
import types
```

```
import sys

def named_tuple(classname, fieldnames):
    # Populate a dictionary of field property accessors
    cls_dict = { name: property(operator.itemgetter(n))
                 for n, name in enumerate(fieldnames) }

    # Make a __new__ function and add to the class dict
    def __new__(cls, *args):
        if len(args) != len(fieldnames):
            raise TypeError('Expected {} arguments'.format(len(fieldnames)))
        return tuple.__new__(cls, args)

    cls_dict['__new__'] = __new__

    # Make the class
    cls = types.new_class(classname, (tuple,), {},
                          lambda ns: ns.update(cls_dict))

    # Set the module to that of the caller
    cls.__module__ = sys._getframe(1).f_globals['__name__']
    return cls
```

The last part of this code uses a so-called "frame hack" involving `sys._getframe()` to obtain the module name of the caller. Another example of frame hacking appears in "Interpolating Variables in Strings".

The following example shows how the preceding code works:

```
>>> Point = named_tuple('Point', ['x', 'y'])
>>> Point
<class '__main__.Point'>
>>> p = Point(4, 5)
>>> len(p)
2
>>> p.x
4
>>> p.y
5
>>> p.x = 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>> print('%s %s' % p)
4 5
>>>
```

One important aspect of the technique used in this recipe is its proper support for metaclasses. You might be inclined to create a class directly by instantiating a metaclass directly. For example:

```
Stock = type('Stock', (), cls_dict)
```

The problem is that this approach skips certain critical steps, such as invocation of the metaclass `__prepare__()` method. By using `types.new_class()` instead, you ensure that all of the necessary initialization steps get carried out. For instance, the callback function that's given as the fourth argument to `types.new_class()` receives the mapping object that's returned by the `__prepare__()` method.

If you only want to carry out the preparation step, use `types.prepare_class()`. For example:

```
import types

metaclass, kwargs, ns = types.prepare_class('Stock', (), {'metaclass': type})
```

This finds the appropriate metaclass and invokes its `__prepare__()` method. The metaclass, remaining keyword arguments, and prepared namespace are then returned.

For more information, see PEP 3115, as well as the Python documentation.

Initializing Class Members at Definition Time

Problem

You want to initialize parts of a class definition once at the time a class is defined, not when instances are created.

Solution

Performing initialization or setup actions at the time of class definition is a classic use of metaclasses. Essentially, a metaclass is triggered at the point of a definition, at which point you can perform additional steps.

Here is an example that uses this idea to create classes similar to named tuples from the `collections` module:

```
import operator

class StructTupleMeta(type):
    def __init__(cls, *args, **kwargs):
        super().__init__(*args, **kwargs)
        for n, name in enumerate(cls._fields):
            setattr(cls, name, property(operator.itemgetter(n)))

class StructTuple(tuple, metaclass=StructTupleMeta):
    _fields = []
    def __new__(cls, *args):
        if len(args) != len(cls._fields):
            raise ValueError('{} arguments required'.format(len(cls._fields)))
        return super().__new__(cls,args)
```

This code allows simple tuple-based data structures to be defined, like this:

```
class Stock(StructTuple):
    _fields = ['name', 'shares', 'price']

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

Here's how they work:

```
>>> s = Stock('ACME', 50, 91.1)
>>> s
('ACME', 50, 91.1)
>>> s[0]
'ACME'
```

```
>>> s.name
'ACME'
>>> s.shares * s.price
4555.0
>>> s.shares = 23
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>>
```

Discussion

In this recipe, the `StructTupleMeta` class takes the listing of attribute names in the `_fields` class attribute and turns them into property methods that access a particular tuple slot. The `operator.itemgetter()` function creates an accessor function and the `property()` function turns it into a property.

The trickiest part of this recipe is knowing when the different initialization steps occur. The `__init__()` method in `StructTupleMeta` is only called once for each class that is defined. The `cls` argument is the class that has just been defined. Essentially, the code is using the `_fields` class variable to take the newly defined class and add some new parts to it.

The `StructTuple` class serves as a common base class for users to inherit from. The `__new__()` method in that class is responsible for making new instances. The use of `__new__()` here is a bit unusual, but is partly related to the fact that we're modifying the calling signature of tuples so that we can create instances with code that uses a normal-looking calling convention like this:

```
s = Stock('ACME', 50, 91.1)        # OK
s = Stock(('ACME', 50, 91.1))      # Error
```

Unlike `__init__()`, the `__new__()` method gets triggered before an instance is created. Since tuples are immutable, it's not possible to make any changes to them once they have been created. An `__init__()` function gets triggered too late in the instance creation process to do what we want. That's why `__new__()` has been defined.

Although this is a short recipe, careful study will reward the reader with a deep insight about how Python classes are defined, how instances are created, and the points at which different methods of metaclasses and classes are invoked.

PEP 422 may provide an alternative means for performing the task described in this recipe. However, as of this writing, it has not been adopted or accepted. Nevertheless, it might be worth a look in case you're working with a version of Python newer than Python 3.3.

Implementing Multiple Dispatch with Function Annotations

Problem

You've learned about function argument annotations and you have a thought that you might be able to use them to implement multiple-dispatch (method overloading) based on types. However, you're not quite sure what's involved (or if it's even a good idea).

Solution

This recipe is based on a simple observation—namely, that since Python allows arguments to be annotated, perhaps it might be possible to write code like this:

```
class Spam:
    def bar(self, x:int, y:int):
        print('Bar 1:', x, y)
    def bar(self, s:str, n:int = 0):
        print('Bar 2:', s, n)

s = Spam()
s.bar(2, 3)     # Prints Bar 1: 2 3
s.bar('hello')  # Prints Bar 2: hello 0
```

Here is the start of a solution that does just that, using a combination of metaclasses and descriptors:

```
# multiple.py

import inspect
import types

class MultiMethod:
    '''
    Represents a single multimethod.
    '''
    def __init__(self, name):
        self._methods = {}
        self.__name__ = name

    def register(self, meth):
        '''
        Register a new method as a multimethod
        '''
        sig = inspect.signature(meth)

        # Build a type signature from the method's annotations
        types = []
        for name, parm in sig.parameters.items():
            if name == 'self':
                continue
            if parm.annotation is inspect.Parameter.empty:
                raise TypeError(
                    'Argument {} must be annotated with a type'.format(name)
                )
            if not isinstance(parm.annotation, type):
                raise TypeError(
                    'Argument {} annotation must be a type'.format(name)
                )
            if parm.default is not inspect.Parameter.empty:
                self._methods[tuple(types)] = meth
            types.append(parm.annotation)

        self._methods[tuple(types)] = meth

    def __call__(self, *args):
        '''
        Call a method based on type signature of the arguments
        '''
```

```
        types = tuple(type(arg) for arg in args[1:])
        meth = self._methods.get(types, None)
        if meth:
            return meth(*args)
        else:
            raise TypeError('No matching method for types {}'.format(types))

    def __get__(self, instance, cls):
        '''
        Descriptor method needed to make calls work in a class
        '''
        if instance is not None:
            return types.MethodType(self, instance)
        else:
            return self

class MultiDict(dict):
    '''
    Special dictionary to build multimethods in a metaclass
    '''
    def __setitem__(self, key, value):
        if key in self:
            # If key already exists, it must be a multimethod or callable
            current_value = self[key]
            if isinstance(current_value, MultiMethod):
                current_value.register(value)
            else:
                mvalue = MultiMethod(key)
                mvalue.register(current_value)
                mvalue.register(value)
                super().__setitem__(key, mvalue)
        else:
            super().__setitem__(key, value)

class MultipleMeta(type):
    '''
    Metaclass that allows multiple dispatch of methods
    '''
    def __new__(cls, clsname, bases, clsdict):
        return type.__new__(cls, clsname, bases, dict(clsdict))

    @classmethod
    def __prepare__(cls, clsname, bases):
        return MultiDict()
```

To use this class, you write code like this:

```
class Spam(metaclass=MultipleMeta):
    def bar(self, x:int, y:int):
        print('Bar 1:', x, y)
    def bar(self, s:str, n:int = 0):
        print('Bar 2:', s, n)

# Example: overloaded __init__
import time
class Date(metaclass=MultipleMeta):
    def __init__(self, year: int, month:int, day:int):
        self.year = year
        self.month = month
        self.day = day
```

```
    def __init__(self):
        t = time.localtime()
        self.__init__(t.tm_year, t.tm_mon, t.tm_mday)
```

Here is an interactive session that verifies that it works:

```
>>> s = Spam()
>>> s.bar(2, 3)
Bar 1: 2 3
>>> s.bar('hello')
Bar 2: hello 0
>>> s.bar('hello', 5)
Bar 2: hello 5
>>> s.bar(2, 'hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "multiple.py", line 42, in __call__
    raise TypeError('No matching method for types {}'.format(types))
TypeError: No matching method for types (<class 'int'>, <class 'str'>)

>>> # Overloaded __init__
>>> d = Date(2012, 12, 21)
>>> # Get today's date
>>> e = Date()
>>> e.year
2012
>>> e.month
12
>>> e.day
3
>>>
```

Discussion

Honestly, there might be too much magic going on in this recipe to make it applicable to real-world code. However, it does dive into some of the inner workings of metaclasses and descriptors, and reinforces some of their concepts. Thus, even though you might not apply this recipe directly, some of its underlying ideas might influence other programming techniques involving metaclasses, descriptors, and function annotations.

The main idea in the implementation is relatively simple. The `MutipleMeta` metaclass uses its `__prepare__()` method to supply a custom class dictionary as an instance of `MultiDict`. Unlike a normal dictionary, `MultiDict` checks to see whether entries already exist when items are set. If so, the duplicate entries get merged together inside an instance of `MultiMethod`.

Instances of `MultiMethod` collect methods by building a mapping from type signatures to functions. During construction, function annotations are used to collect these signatures and build the mapping. This takes place in the `MultiMethod.register()` method. One critical part of this mapping is that for multimethods, types must be specified on all of the arguments or else an error occurs.

To make `MultiMethod` instances emulate a callable, the `__call__()` method is implemented. This method builds a type tuple from all of the arguments except `self`, looks up the method in the internal map, and invokes the appropriate method. The `__get__()` is required to make `MultiMethod` instances operate correctly inside class definitions. In the implementation, it's being used to create proper bound methods. For example:

```
>>> b = s.bar
>>> b
<bound method Spam.bar of <__main__.Spam object at 0x1006a46d0>>
>>> b.__self__
<__main__.Spam object at 0x1006a46d0>
>>> b.__func__
<__main__.MultiMethod object at 0x1006a4d50>
>>> b(2, 3)
Bar 1: 2 3
>>> b('hello')
Bar 2: hello 0
>>>
```

To be sure, there are a lot of moving parts to this recipe. However, it's all a little unfortunate considering how many limitations there are. For one, the solution doesn't work with keyword arguments: For example:

```
>>> s.bar(x=2, y=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __call__() got an unexpected keyword argument 'y'

>>> s.bar(s='hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __call__() got an unexpected keyword argument 's'
>>>
```

There might be some way to add such support, but it would require a completely different approach to method mapping. The problem is that the keyword arguments don't arrive in any kind of particular order. When mixed up with positional arguments, you simply get a jumbled mess of arguments that you have to somehow sort out in the `__call__()` method.

This recipe is also severely limited in its support for inheritance. For example, something like this doesn't work:

```
class A:
    pass

class B(A):
    pass

class C:
    pass

class Spam(metaclass=MultipleMeta):
    def foo(self, x:A):
        print('Foo 1:', x)

    def foo(self, x:C):
        print('Foo 2:', x)
```

The reason it fails is that the `x:A` annotation fails to match instances that are subclasses (such as instances of `B`). For example:

```
>>> s = Spam()
>>> a = A()
>>> s.foo(a)
```

```
Foo 1: <__main__.A object at 0x1006a5310>
>>> c = C()
>>> s.foo(c)
Foo 2: <__main__.C object at 0x1007a1910>
>>> b = B()
>>> s.foo(b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "multiple.py", line 44, in __call__
    raise TypeError('No matching method for types {}'.format(types))
TypeError: No matching method for types (<class '__main__.B'>,)
>>>
```

As an alternative to using metaclasses and annotations, it is possible to implement a similar recipe using decorators. For example:

```
import types

class multimethod:
    def __init__(self, func):
        self._methods = {}
        self.__name__ = func.__name__
        self._default = func

    def match(self, *types):
        def register(func):
            ndefaults = len(func.__defaults__) if func.__defaults__ else 0
            for n in range(ndefaults+1):
                self._methods[types[:len(types) - n]] = func
            return self
        return register

    def __call__(self, *args):
        types = tuple(type(arg) for arg in args[1:])
        meth = self._methods.get(types, None)
        if meth:
            return meth(*args)
        else:
            return self._default(*args)

    def __get__(self, instance, cls):
        if instance is not None:
            return types.MethodType(self, instance)
        else:
            return self
```

To use the decorator version, you would write code like this:

```
class Spam:
    @multimethod
    def bar(self, *args):
        # Default method called if no match
        raise TypeError('No matching method for bar')

    @bar.match(int, int)
    def bar(self, x, y):
        print('Bar 1:', x, y)

    @bar.match(str, int)
    def bar(self, s, n = 0):
```

```
        print('Bar 2:', s, n)
```

The decorator solution also suffers the same limitations as the previous implementation (namely, no support for keyword arguments and broken inheritance).

All things being equal, it's probably best to stay away from multiple dispatch in general-purpose code. There are special situations where it might make sense, such as in programs that are dispatching methods based on some kind of pattern matching. For example, perhaps the visitor pattern described in "Implementing the Visitor Pattern" could be recast into a class that used multiple dispatch in some way. However, other than that, it's usually never a bad idea to stick with a more simple approach (simply use methods with different names).

Ideas concerning different ways to implement multiple dispatch have floated around the Python community for years. As a decent starting point for that discussion, see Guido van Rossum's blog post "Five-Minute Multimethods in Python".

Avoiding Repetitive Property Methods

Problem

You are writing classes where you are repeatedly having to define property methods that perform common tasks, such as type checking. You would like to simplify the code so there is not so much code repetition.

Solution

Consider a simple class where attributes are being wrapped by property methods:

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

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, value):
        if not isinstance(value, str):
            raise TypeError('name must be a string')
        self._name = value

    @property
    def age(self):
        return self._age

    @age.setter
    def age(self, value):
        if not isinstance(value, int):
            raise TypeError('age must be an int')
        self._age = value
```

As you can see, a lot of code is being written simply to enforce some type assertions on attribute values. Whenever you see code like this, you should explore different ways of simplifying it. One possible

approach is to make a function that simply defines the property for you and returns it. For example:

```
def typed_property(name, expected_type):
    storage_name = '_' + name

    @property
    def prop(self):
        return getattr(self, storage_name)

    @prop.setter
    def prop(self, value):
        if not isinstance(value, expected_type):
            raise TypeError('{} must be a {}'.format(name, expected_type))
        setattr(self, storage_name, value)
    return prop

# Example use
class Person:
    name = typed_property('name', str)
    age = typed_property('age', int)
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Discussion

This recipe illustrates an important feature of inner function or closures—namely, their use in writing code that works a lot like a macro. The `typed_property()` function in this example may look a little weird, but it's really just generating the property code for you and returning the resulting property object. Thus, when it's used in a class, it operates exactly as if the code appearing inside `typed_property()` was placed into the class definition itself. Even though the property getter and setter methods are accessing local variables such as `name`, `expected_type`, and `storage_name`, that is fine—those values are held behind the scenes in a closure.

This recipe can be tweaked in an interesting manner using the `functools.partial()` function. For example, you can do this:

```
from functools import partial

String = partial(typed_property, expected_type=str)
Integer = partial(typed_property, expected_type=int)

# Example:
class Person:
    name = String('name')
    age = Integer('age')
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Here the code is starting to look a lot like some of the type system descriptor code shown in "Implementing a Data Model or Type System".

Defining Context Managers the Easy Way

Problem

You want to implement new kinds of context managers for use with the `with` statement.

Solution

One of the most straightforward ways to write a new context manager is to use the `@contextmanager` decorator in the `contextlib` module. Here is an example of a context manager that times the execution of a code block:

```
import time
from contextlib import contextmanager

@contextmanager
def timethis(label):
    start = time.time()
    try:
        yield
    finally:
        end = time.time()
        print('{}: {}'.format(label, end - start))

# Example use
with timethis('counting'):
    n = 10000000
    while n > 0:
        n -= 1
```

In the `timethis()` function, all of the code prior to the `yield` executes as the `__enter__()` method of a context manager. All of the code after the `yield` executes as the `__exit__()` method. If there was an exception, it is raised at the `yield` statement.

Here is a slightly more advanced context manager that implements a kind of transaction on a list object:

```
@contextmanager
def list_transaction(orig_list):
    working = list(orig_list)
    yield working
    orig_list[:] = working
```

The idea here is that changes made to a list only take effect if an entire code block runs to completion with no exceptions. Here is an example that illustrates:

```
>>> items = [1, 2, 3]
>>> with list_transaction(items) as working:
...     working.append(4)
...     working.append(5)
...
>>> items
[1, 2, 3, 4, 5]
>>> with list_transaction(items) as working:
...     working.append(6)
...     working.append(7)
...     raise RuntimeError('oops')
...
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
```

```
RuntimeError: oops
>>> items
[1, 2, 3, 4, 5]
>>>
```

Discussion

Normally, to write a context manager, you define a class with an `__enter__()` and `__exit__()` method, like this:

```
import time

class timethis:
    def __init__(self, label):
        self.label = label
    def __enter__(self):
        self.start = time.time()
    def __exit__(self, exc_ty, exc_val, exc_tb):
        end = time.time()
        print('{}: {}'.format(self.label, end - self.start))
```

Although this isn't hard, it's a lot more tedious than writing a simple function using `@contextmanager`.

`@contextmanager` is really only used for writing self-contained context-management functions. If you have some object (e.g., a file, network connection, or lock) that needs to support the `with` statement, you still need to implement the `__enter__()` and `__exit__()` methods separately.

Executing Code with Local Side Effects

Problem

You are using `exec()` to execute a fragment of code in the scope of the caller, but after execution, none of its results seem to be visible.

Solution

To better understand the problem, try a little experiment. First, execute a fragment of code in the global namespace:

```
>>> a = 13
>>> exec('b = a + 1')
>>> print(b)
14
>>>
```

Now, try the same experiment inside a function:

```
>>> def test():
...     a = 13
...     exec('b = a + 1')
...     print(b)
...
>>> test()
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in test
NameError: global name 'b' is not defined
>>>
```

As you can see, it fails with a `NameError` almost as if the `exec()` statement never actually executed. This can be a problem if you ever want to use the result of the `exec()` in a later calculation.

To fix this kind of problem, you need to use the `locals()` function to obtain a dictionary of the local variables prior to the call to `exec()`. Immediately afterward, you can extract modified values from the locals dictionary. For example:

```
>>> def test():
...     a = 13
...     loc = locals()
...     exec('b = a + 1')
...     b = loc['b']
...     print(b)
...
>>> test()
14
>>>
```

Discussion

Correct use of `exec()` is actually quite tricky in practice. In fact, in most situations where you might be considering the use of `exec()`, a more elegant solution probably exists (e.g., decorators, closures, metaclasses, etc.).

However, if you still must use `exec()`, this recipe outlines some subtle aspects of using it correctly. By default, `exec()` executes code in the local and global scope of the caller. However, inside functions, the local scope passed to `exec()` is a dictionary that is a copy of the actual local variables. Thus, if the code in `exec()` makes any kind of modification, that modification is never reflected in the actual local variables. Here is another example that shows this effect:

```
>>> def test1():
...     x = 0
...     exec('x += 1')
...     print(x)
...
>>> test1()
0
>>>
```

When you call `locals()` to obtain the local variables, as shown in the solution, you get the copy of the locals that is passed to `exec()`. By inspecting the value of the dictionary after execution, you can obtain the modified values. Here is an experiment that shows this:

```
>>> def test2():
...     x = 0
...     loc = locals()
...     print('before:', loc)
...     exec('x += 1')
...     print('after:', loc)
...     print('x =', x)
...
```

```
>>> test2()
before: {'x': 0}
after: {'loc': {...}, 'x': 1}
x = 0
>>>
```

Carefully observe the output of the last step. Unless you copy the modified value from `loc` back to `x`, the variable remains unchanged.

With any use of `locals()`, you need to be careful about the order of operations. Each time it is invoked, `locals()` will take the current value of local variables and overwrite the corresponding entries in the dictionary. Observe the outcome of this experiment:

```
>>> def test3():
...     x = 0
...     loc = locals()
...     print(loc)
...     exec('x += 1')
...     print(loc)
...     locals()
...     print(loc)
...
>>> test3()
{'x': 0}
{'loc': {...}, 'x': 1}
{'loc': {...}, 'x': 0}
>>>
```

Notice how the last call to `locals()` caused `x` to be overwritten.

As an alternative to using `locals()`, you might make your own dictionary and pass it to `exec()`. For example:

```
>>> def test4():
...     a = 13
...     loc = { 'a' : a }
...     glb = { }
...     exec('b = a + 1', glb, loc)
...     b = loc['b']
...     print(b)
...
>>> test4()
14
>>>
```

For most uses of `exec()`, this is probably good practice. You just need to make sure that the global and local dictionaries are properly initialized with names that the executed code will access.

Last, but not least, before using `exec()`, you might ask yourself if other alternatives are available. Many problems where you might consider the use of `exec()` can be replaced by closures, decorators, metaclasses, or other metaprogramming features.

Parsing and Analyzing Python Source

Problem

You want to write programs that parse and analyze Python source code.

Solution

Most programmers know that Python can evaluate or execute source code provided in the form of a string. For example:

```
>>> x = 42
>>> eval('2 + 3*4 + x')
56
>>> exec('for i in range(10): print(i)')
0
1
2
3
4
5
6
7
8
9
>>>
```

However, the `ast` module can be used to compile Python source code into an abstract syntax tree (AST) that can be analyzed. For example:

```
>>> import ast
>>> ex = ast.parse('2 + 3*4 + x', mode='eval')
>>> ex
<_ast.Expression object at 0x1007473d0>
>>> ast.dump(ex)
"Expression(body=BinOp(left=BinOp(left=Num(n=2), op=Add(),
right=BinOp(left=Num(n=3), op=Mult(), right=Num(n=4))), op=Add(),
right=Name(id='x', ctx=Load())))"

>>> top = ast.parse('for i in range(10): print(i)', mode='exec')
>>> top
<_ast.Module object at 0x100747390>
>>> ast.dump(top)
"Module(body=[For(target=Name(id='i', ctx=Store()),
iter=Call(func=Name(id='range', ctx=Load()), args=[Num(n=10)],
keywords=[], starargs=None, kwargs=None),
body=[Expr(value=Call(func=Name(id='print', ctx=Load()),
args=[Name(id='i', ctx=Load())], keywords=[], starargs=None,
kwargs=None))], orelse=[])])"
>>>
```

Analyzing the source tree requires a bit of study on your part, but it consists of a collection of AST nodes. The easiest way to work with these nodes is to define a visitor class that implements various `visit_NodeName()` methods where `NodeName()` matches the node of interest. Here is an example of such a class that records information about which names are loaded, stored, and deleted.

```
import ast

class CodeAnalyzer(ast.NodeVisitor):
    def __init__(self):
        self.loaded = set()
        self.stored = set()
```

```
        self.deleted = set()
    def visit_Name(self, node):
        if isinstance(node.ctx, ast.Load):
            self.loaded.add(node.id)
        elif isinstance(node.ctx, ast.Store):
            self.stored.add(node.id)
        elif isinstance(node.ctx, ast.Del):
            self.deleted.add(node.id)

# Sample usage
if __name__ == '__main__':
    # Some Python code
    code = '''
for i in range(10):
    print(i)
del i
'''
    # Parse into an AST
    top = ast.parse(code, mode='exec')

    # Feed the AST to analyze name usage
    c = CodeAnalyzer()
    c.visit(top)
    print('Loaded:', c.loaded)
    print('Stored:', c.stored)
    print('Deleted:', c.deleted)
```

If you run this program, you'll get output like this:

```
Loaded: {'i', 'range', 'print'}
Stored: {'i'}
Deleted: {'i'}
```

Finally, ASTs can be compiled and executed using the `compile()` function. For example:

```
>>> exec(compile(top,'<stdin>', 'exec'))
0
1
2
3
4
5
6
7
8
9
>>>
```

Discussion

The fact that you can analyze source code and get information from it could be the start of writing various code analysis, optimization, or verification tools. For instance, instead of just blindly passing some fragment of code into a function like `exec()`, you could turn it into an AST first and look at it in some detail to see what it's doing. You could also write tools that look at the entire source code for a module and perform some sort of static analysis over it.

It should be noted that it is also possible to rewrite the AST to represent new code if you really know what you're doing. Here is an example of a decorator that lowers globally accessed names into the body of a

function by reparsing the function body's source code, rewriting the AST, and recreating the function's code object:

```
# namelower.py
import ast
import inspect

# Node visitor that lowers globally accessed names into
# the function body as local variables.
class NameLower(ast.NodeVisitor):
    def __init__(self, lowered_names):
        self.lowered_names = lowered_names

    def visit_FunctionDef(self, node):
        # Compile some assignments to lower the constants
        code = '__globals = globals()\n'
        code += '\n'.join("{0} = __globals['{0}']".format(name)
                          for name in self.lowered_names)

        code_ast = ast.parse(code, mode='exec')

        # Inject new statements into the function body
        node.body[:0] = code_ast.body

        # Save the function object
        self.func = node

# Decorator that turns global names into locals
def lower_names(*namelist):
    def lower(func):
        srclines = inspect.getsource(func).splitlines()
        # Skip source lines prior to the @lower_names decorator
        for n, line in enumerate(srclines):
            if '@lower_names' in line:
                break

        src = '\n'.join(srclines[n+1:])
        # Hack to deal with indented code
        if src.startswith((' ','\t')):
            src = 'if 1:\n' + src
        top = ast.parse(src, mode='exec')

        # Transform the AST
        cl = NameLower(namelist)
        cl.visit(top)

        # Execute the modified AST
        temp = {}
        exec(compile(top,'','exec'), temp, temp)

        # Pull out the modified code object
        func.__code__ = temp[func.__name__].__code__
        return func
    return lower
```

To use this code, you would write code such as the following:

```
INCR = 1

@lower_names('INCR')
```

```
def countdown(n):
    while n > 0:
        n -= INCR
```

The decorator rewrites the source code of the `countdown()` function to look like this:

```
def countdown(n):
    __globals = globals()
    INCR = __globals['INCR']
    while n > 0:
        n -= INCR
```

In a performance test, it makes the function run about 20% faster.

Now, should you go applying this decorator to all of your functions? Probably not. However, it's a good illustration of some very advanced things that might be possible through AST manipulation, source code manipulation, and other techniques.

This recipe was inspired by a similar recipe at ActiveState that worked by manipulating Python's byte code. Working with the AST is a higher-level approach that might be a bit more straightforward. See the next recipe for more information about byte code.

Disassembling Python Byte Code

Problem

You want to know in detail what your code is doing under the covers by disassembling it into lower-level byte code used by the interpreter.

Solution

The `dis` module can be used to output a disassembly of any Python function. For example:

```
>>> def countdown(n):
...     while n > 0:
...         print('T-minus', n)
...         n -= 1
...     print('Blastoff!')
...
>>> import dis
>>> dis.dis(countdown)
  2           0 SETUP_LOOP              39 (to 42)
        >>    3 LOAD_FAST                0 (n)
              6 LOAD_CONST               1 (0)
              9 COMPARE_OP               4 (>)
             12 POP_JUMP_IF_FALSE       41

  3          15 LOAD_GLOBAL              0 (print)
             18 LOAD_CONST               2 ('T-minus')
             21 LOAD_FAST                0 (n)
             24 CALL_FUNCTION            2 (2 positional, 0 keyword pair)
             27 POP_TOP

  4          28 LOAD_FAST                0 (n)
             31 LOAD_CONST               3 (1)
```

```
              34 INPLACE_SUBTRACT
              35 STORE_FAST               0 (n)
              38 JUMP_ABSOLUTE            3
        >>    41 POP_BLOCK

  5     >>    42 LOAD_GLOBAL              0 (print)
              45 LOAD_CONST               4 ('Blastoff!')
              48 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
              51 POP_TOP
              52 LOAD_CONST               0 (None)
              55 RETURN_VALUE
>>>
```

Discussion

The `dis` module can be useful if you ever need to study what's happening in your program at a very low level (e.g., if you're trying to understand performance characteristics).

The raw byte code interpreted by the `dis()` function is available on functions as follows:

```
>>> countdown.__code__.co_code
b"x'\x00|\x00\x00d\x01\x00k\x04\x00r)\x00t\x00\x00d\x02\x00|\x00\x00\x83
\x02\x00\x01|\x00\x00d\x03\x008}\x00\x00q\x03\x00Wt\x00\x00d\x04\x00\x83
\x01\x00\x01d\x00\x00S"
>>>
```

If you ever want to interpret this code yourself, you would need to use some of the constants defined in the `opcode` module. For example:

```
>>> c = countdown.__code__.co_code
>>> import opcode
>>> opcode.opname[c[0]]
>>> opcode.opname[c[0]]
'SETUP_LOOP'
>>> opcode.opname[c[3]]
'LOAD_FAST'
>>>
```

Ironically, there is no function in the `dis` module that makes it easy for you to process the byte code in a programmatic way. However, this generator function will take the raw byte code sequence and turn it into opcodes and arguments.

```
import opcode

def generate_opcodes(codebytes):
    extended_arg = 0
    i = 0
    n = len(codebytes)
    while i < n:
        op = codebytes[i]
        i += 1
        if op >= opcode.HAVE_ARGUMENT:
            oparg = codebytes[i] + codebytes[i+1]*256 + extended_arg
            extended_arg = 0
            i += 2
            if op == opcode.EXTENDED_ARG:
                extended_arg = oparg * 65536
                continue
```

```
        else:
            oparg = None
        yield (op, oparg)
```

To use this function, you would use code like this:

```
>>> for op, oparg in generate_opcodes(countdown.__code__.co_code):
...     print(op, opcode.opname[op], oparg)
...
120 SETUP_LOOP 39
124 LOAD_FAST 0
100 LOAD_CONST 1
107 COMPARE_OP 4
114 POP_JUMP_IF_FALSE 41
116 LOAD_GLOBAL 0
100 LOAD_CONST 2
124 LOAD_FAST 0
131 CALL_FUNCTION 2
1 POP_TOP None
124 LOAD_FAST 0
100 LOAD_CONST 3
56 INPLACE_SUBTRACT None
125 STORE_FAST 0
113 JUMP_ABSOLUTE 3
87 POP_BLOCK None
116 LOAD_GLOBAL 0
100 LOAD_CONST 4
131 CALL_FUNCTION 1
1 POP_TOP None
100 LOAD_CONST 0
83 RETURN_VALUE None
>>>
```

It's a little-known fact, but you can replace the raw byte code of any function that you want. It takes a bit of work to do it, but here's an example of what's involved:

```
>>> def add(x, y):
...     return x + y
...
>>> c = add.__code__
>>> c
<code object add at 0x1007beed0, file "<stdin>", line 1>
>>> c.co_code
b'|\x00\x00|\x01\x00\x17S'
>>>
>>> # Make a completely new code object with bogus byte code
>>> import types
>>> newbytecode = b'xxxxxxx'
>>> nc = types.CodeType(c.co_argcount, c.co_kwonlyargcount,
...    c.co_nlocals, c.co_stacksize, c.co_flags, newbytecode, c.co_consts,
...    c.co_names, c.co_varnames, c.co_filename, c.co_name,
...    c.co_firstlineno, c.co_lnotab)
>>> nc
<code object add at 0x10069fe40, file "<stdin>", line 1>
>>> add.__code__ = nc
>>> add(2,3)
Segmentation fault
```

Having the interpreter crash is a pretty likely outcome of pulling a crazy stunt like this. However,

developers working on advanced optimization and metaprogramming tools might be inclined to rewrite byte code for real. This last part illustrates how to do it. See this code on ActiveState for another example of such code in action.

Prev Next
Chapter 8. Classes and Objects Home Chapter 10. Modules and Packages