Python Programming Tips #2

Spreading a Class Over Multiple Files

It is expected that any class implemented in Python will occupy a single module. So, for example, if we have a class called `DataStore`, by convention we would put it in a module called `DataStore.py`. To access the class we would do `import DataStore` and then create an instance using code like `store = DataStore.DataStore(...)` where the ellipsis stands for any arguments we might pass.

But what happens if our class has some very large methods? Or lots or methods? Or both? Personally, I prefer to keep my modules under 1,000 lines, but larger classes—such as those that provide an application's data storage or a top-level window in a GUI application—can easily run to well over 1,000 lines. Fortunately, thanks to Python's flexibility and meta-programming support (don't worry, the code is short and straightforward!) we can keep our modules to a managable size by spreading a class over as many files as we like.

Probably the most common approach to spreading a class's code over multiple files is to use subclassing, with the base class in one module, and the (or each) subclass, in its own separate module. This works fine when there is a logical division of functionality into a base class/subclass hierarchy, but isn't suitable for cases where all the functionality needs to be at the same level, which is the case this article addresses.

We'll start with our “large” module, and then look at a very simple way to spread its code over multiple files. This first way has some disadvantages, so we'll then show a second way that uses a tiny bit of meta-programming, then a third way that is what I actually used to use in practice, and finally the “definitive” technique I now use. (Skip to the definitive way.)

First though, our “large” module:

```
# DataStore.py

class DataStore:

    ... # lots of other methods ...

    def bigMethod(self):
        ... # lots of lines

    def hugeMethod(self):
        ... # lots of lines
```

Overall, this module is too big and we'd rather split it up. A simple solution is to delegate the work to functions in a separate module. For example:

```
# DataStore.py
import _DataStore

class DataStore:

    ... # lots of other methods ...

    def bigMethod(self):
        state = ... # accumulate all the necessary state
```

```
        _DataStore.bigMethod(state)

    def hugeMethod(self):
        state = ... # accumulate all the necessary state
        _DataStore.hugeMethod(state)


# _DataStore.py

def bigMethod(state):
    ... # lots of lines

def hugeMethod(state):
    ... # lots of lines
```

I've called the supporting module `_DataStore.py` to indicate that is a private version of `DataStore.py`, but that's just my personal convention. (When I spread a class over multiple files I add more description, for example, `_DataStoreSelect.py`, `_DataStoreModify.py`, etc.)

Unfortunately, there are a couple of disadvantages to this approach. Firstly, every method call to a delegated method becomes a method call plus a function call. The overhead of this should be insignificant though, because we normally use the technique for big methods anyway. (But it does preclude our use of small or fast methods that are used a lot.) Also, we have to pass the instance's state to give the function access to the class's methods and the instance's data.

```
# DataStore.py
import _DataStore

class DataStore:

    ... # lots of other methods ...

    def bigMethod(self):
        _DataStore.bigMethod(self)

    def hugeMethod(self):
        _DataStore.hugeMethod(self)


# _DataStore.py

# self is a DataStore

def bigMethod(self):
    ... # lots of lines

def hugeMethod(self):
    ... # lots of lines
```

Here we've solved the problem of passing state by simply passing `self` to the functions. But we still have the delegation—it would be nice to get rid of that, and in fact we can do so by using a little bit of meta-programming.

```
# DataStore.py
import Lib
import _DataStore
```

```
@Lib.add_functions_as_methods(
    (_DataStore.bigMethod, _DataStore.hugeMethod))
class DataStore:

    ... # lots of other methods ...


# Lib.py

def add_functions_as_methods(functions):
    def decorator(Class):
        for function in functions:
            setattr(Class, function.__name__, function)
        return Class
    return decorator
```

Here we've used a class decorator that given a sequence of functions, adds each one to the decorated class as a method. So now, there's no need for delegation and our `DataStore` class has its `bigMethod()` and `hugeMethod()` methods even though they are implemented in a separate `_DataStore.py` module.

I don't like having to name all the functions that are to become methods when I call the class decorator, so my convention is to create a `functions` variable at the end of the private module(s) that lists the functions to be used by the class decorator.

```
# DataStore.py
import Lib
import _DataStore

@Lib.add_functions_as_methods(_DataStore.functions)
class DataStore:

    ... # lots of other methods ...


# _DataStore.py

# self is a DataStore

def bigMethod(self):
    ... # lots of lines

def hugeMethod(self):
    ... # lots of lines

functions = (bigMethod, hugeMethod)
```

Of course, it would be nicer if we didn't have to remember to populate the `functions` list at the end. Also it would be nice to be able to distinguish between functions that are to be added to a class as methods and their helpers. Here is my original code which works, but is rather simple and slightly inconvenient.

```
# Lib.py

def add_functions_as_methods(functions):
```

```
    def decorator(Class):
        for function in functions:
            setattr(Class, function.__name__, function)
        return Class
    return decorator

def register_function(sequence, function):
    sequence.append(function)
    return function


# _DataStore.py

import functools
import Lib

functions = []
register_function = functools.partial(
    Lib.register_function, functions)

# self is a DataStore

@register_function
def bigMethod(self):
    ... # lots of lines

@register_function
def hugeMethod(self):
    ... # lots of lines
```

Here, I've added a new function to the `Lib` module that is used as a decorator and modified the private `_DataStore` module to make use of it. Rather unusually, instead of returning a modified function, the decorator returns the original function unchanged, but as a side-effect, adds the function to the given sequence—in this case the `functions` list. The imports and two extra statements at the start of the private `_DataStore` are boilerplate that can be copied verbatim; then all that's needed is to use the `@register_function` decorator for any functions that are ultimately to be added as methods. (Incidentally, the public `DataStore` module itself requires no changes.)

Although the example only spreads the class over two files, the technique can easily be used to spread a class over as many files as required. Each supplementary module should have the boilerplate code near the start, i.e.,

```
import functools
import Lib

functions = []
register_function = functools.partial(
    Lib.register_function, functions)
```

In addition, each function to be used as a method will need the `@register_function` decorator, and of course, for each of these modules the main module needs an extra `@Lib.add_functions_as_methods(`*`_supplementaryModule`*`.functions)` statement to be added in front of the `class` definition.

Here's what used to be my definitive version.

Since originally writing this article I've used this technique quite a lot, and I've now refined and simplified it. First up are the improved library functions, then we'll see them in use.

```
# Lib.py

def add_methods_from(*modules):
    def decorator(Class):
        for module in modules:
            for method in getattr(module, "__methods__"):
                setattr(Class, method.__name__, method)
        return Class
    return decorator

def register_method(methods):
    def register_method(method):
        methods.append(method)
        return method # Unchanged
    return register_method
```

The `add_methods_from()` function takes any number of modules and for each one adds any methods that have been registered with the class it is used to decorate.

```
# DataStore.py

import Lib
import _DataStore

@Lib.add_methods_from(_DataStore) # Don't need to specify which methods.
class DataStore:                  # Can pass multiple modules.

    def __init__(self):
        self._a = 1
        self._b = 2
        self._c = 3

    def small_method(self):
        return self._a
```

We can have as many `add_methods_from()` decorators as we like, or we can just use one with multiple modules, e.g., `@Lib.add_methods_from(_DataStoreLoad, _DataStoreSave)`.

```
# _DataStore

import Lib

__methods__ = [] # self is a DataStore
register_method = Lib.register_method(__methods__)

@register_method
def big_method(self):
    return self._b

@register_method
```

```
def huge_method(self):
    return self._c
```

We *must* provide a `__methods__` list since the `Lib.add_methods_from()` function depends on it. Notice that we don't use partial function application any more (in fact, it wasn't needed before, but I hadn't noticed at the time).

The Definitive Version?

Thanks to Garry Herron's post on the comp.lang.python newsgroup, I now use a much simpler, cleaner approach: mixins. These are classes that have no data of their own—only methods—so although you inherit them, you never have to call `super()` on them, and they "just work". Nor do we need any library functions, since Python supports this out of the box.

```
# DataStore.py

import _DataStore

class DataStore(_DataStore.Mixin): # Could inherit many more mixins

    def __init__(self):
        self._a = 1
        self._b = 2
        self._c = 3

    def small_method(self):
        return self._a
```

Our mixin classes must not have an `__init__` or store any data—but they have full access to `self` and *its* data of course.

```
# _DataStore.py

class Mixin:

    def big_method(self):
        return self._b

    def huge_method(self):
        return self._c
```

Simpler, and nicer—and pure Python. However, some posters on the newsgroup felt that other approaches would be better, e.g., Steven D'Aprano's reply importing into a class. In fact, nowadays, I tend to just use pure delegation.

For more see Python Programming Tips
