Component Adaptation + Open Protocols = The PyProtocols Package

Release 1.0a0

Phillip J. Eby

October 10, 2004

Email: pje@telecommunity.com

Abstract

The Python Protocols package provides framework developers and users with tools for defining, declaring, and adapting components between interfaces, even when those interfaces are defined using different mechanisms.

CONTENTS

CHAPTER

ONE

Reference

1.1 `protocols` — Protocol Definition, Declaration, and Adaptation

> The typical Python programmer is an integrator, someone who is connecting components from various vendors. Often times the interfaces between these components require an intermediate adapter. Usually the burden falls upon the programmer to study the interface exposed by one component and required by another, determine if they are directly compatible, or develop an adapter. Sometimes a vendor may even include the appropriate adapter, but then searching for the adapter and figuring out how to deploy the adapter takes time.
>
> — Martelli & Evans, PEP 246

This package builds on the object adaptation protocol presented in PEP 246 to make it easier for component authors, framework suppliers, and other developers to:

- Specify what behavior a component requires or provides
- Specify how to adapt the interface provided by one component to that required by another
- Specify how to adapt objects of a particular type or class (even built-in types) to a particular required interface
- Automatically adapt a supplied object to a required interface, and
- Do all of the above, even when the components or frameworks involved were not written to take advantage of this package, and even if the frameworks have different mechanisms for defining interfaces.

Assuming that a particular framework either already supports this package, or has been externally adapted to do so, then framework users will typically use this package's declaration API to declare what interfaces their classes or objects provide, and/or to declare adaptations between interfaces or components.

For framework developers, this package offers an opportunity to replace tedious and repetitive type-checking code (such as `isinstance()`, `type()`, `hasattr()`, or interface checks) with single calls to `adapt()` instead. In addition, if the framework has objects that represent interfaces or protocols, the framework developer can make them usable with this package's declaration API by adding adapters for (or direct implementations of) the `IOpenProtocol` interface provided herein.

If the developer of a framework does not do these things, it may still be possible for a framework user or third-party developer to do them, in order to be able to use this package's API. The user of a framework can often call `adapt()` on a component before passing it to a non-adapting framework. And, it's possible to externally adapt a framework's interface objects as well.

For example, the `protocols.zope_support` and `protocols.twisted_support` modules define adapters that implement `IOpenProtocol` on behalf of Zope and Twisted `Interface` objects. This allows them to be used as arguments to this package's protocol declaration API. This works even though Zope and Twisted are completely

unaware of the `protocols` package. (Of course, this does not give Zope or Twisted `Interface` objects all of the capabilities that `Protocol` objects have, but it does make most of their existing functionality accessible through the same API.)

Finally, framework and non-framework developers alike may also wish to use the `Protocol` and `Interface` base classes from this package to define protocols or interfaces of their own, or perhaps use some of the adaptation mechanisms supplied here to implement "double dispatching" or the "visitor pattern".

See Also:

PEP 246, "*Object Adaptation*"
: PEP 246 describes an early version of the adaptation protocol used by this package.

1.1.1 Big Example 1 — A Python Documentation Framework

To provide the reader with a "feel" for the use and usefulness of the `protocols` package, we will begin with a motivating example: a simple Python documentation framework. To avoid getting bogged down in details, we will only sketch a skeleton of the framework, highlighting areas where the `protocols` package would play a part.

First, let's consider the background and requirements. Python has many documentation tools available, ranging from the built-in **pydoc** to third-party tools such as **HappyDoc**, **epydoc**, and **Synopsis**. Many of these tools generate documentation from "live" Python objects, some using the Python `inspect` module to do so.

However, such tools often encounter difficulties in the Python 2.2 world. Tools that use `type()` checks break with custom metaclasses, and even tools that use `isinstance()` break when dealing with custom descriptor types. These tools often handle other custom object types poorly as well: for example, Zope `Interface` objects can cause some versions of `help()` and **pydoc** to crash!

The state of Python documentation tools is an example of the problem that both PEP 246 and the `protocols` package were intended to solve: introspection makes frameworks brittle and unextensible. We can't easily plug new kinds of "documentables" into our documentation tools, or control how existing objects are documented. These are exactly the kind of problems that component adaptation and open protocols were created to address.

So let's review our requirements for the documentation framework. First, it should work with existing built-in types, without requiring a new version of Python. Second, it should allow users to control how objects are recognized and documented. Third, we want the framework to be flexible enough to create different kinds of documentation, like JavaDoc-style HTML, PDF reference manuals, plaintext online help or manpages, and so on – with whatever kinds of documentable objects exist. If user A creates a new "documentable object," and user B creates a new documentation format, user C should be able to combine the two.

To design a framework with the `protocols` package, the best place to start is often with an "ideal" interface. We pretend that every object is already the kind of object that would do everything we need it to do. In the case of documentation, we want objects to be able to tell us their name, what kind of object they should be listed as, their base classes (if applicable), their call signature (if callable), and their contents (if a namespace).

So let's define our ideal interface, using `protocols.Interface`. (**Note:** This is not the only way to define an interface; we could also use an abstract base class, or other techniques. Interface definition is discussed further in section 1.1.4.)

```
from protocols import Interface

class IDocumentable(Interface):
    """A documentable object"""

    def getKind():
        """Return "kind" string, e.g. "class", "interface", "package", etc."""

    def getName():
        """Return an unqualified (undotted) name of the object"""

    def getBases():
        """Return a sequence of objects this object is derived from
        (or an empty sequence if object is not class/type-like)"""

    def getSignature():
        """Return a description of the object's call signature, or None"""

    def getSummary():
        """Return a one-line summary description of the object"""

    def getDoc():
        """Return documentation for the object (not including its contents)"""

    def getContents(includeInherited):
        """Return list of (key,value) pairs for namespace contents, if any"""
```

Now, in the "real world," no existing Python objects provide this interface. But, if every Python object did, we'd be in great shape for writing documentation tools. The tools could focus purely on issues of formatting and organization, not "digging up dirt" on what to say about the objects.

Notice that we don't need to care whether an object is a type or a module or a staticmethod or something else. If a future version of Python made modules callable or functions able to somehow inherit from other functions, we'd be covered, as long as those new objects supported the methods we described. Even if a tool needs to care about the "package" kind vs. the "module" kind for formatting or organizational reasons, it can easily be written to assume that new "kinds" might still appear. For example, an index generator might just generate separate alphabetical indexes for each "kind" it encounters during processing.

Okay, so we've envisioned our ideal scenario, and documented it as an interface. Now what? Well, we could start writing documentation tools that expect to be given objects that support the `IDocumentable` interface, but that wouldn't be very useful, since we don't have any `IDocumentable` objects yet. So let's define some **adapters** for built-in types, so that we have something for our tools to document.

```
from protocols import advise
from types import FunctionType

class FunctionAsDocumentable(object):

    advise(
        # Declare that this class provides IDocumentable for FunctionType
        # (we'll explain this further in later sections)

        provides = [IDocumentable],
        asAdapterForTypes = [FunctionType],
    )

    def __init__(self, ob):
        self.func = ob

    def getKind(self):
        return "function"

    def getName(self):
        return self.func.func_name

    def getBases(self):
        return ()

    # ... etc.
```

The `FunctionAsDocumentable` class wraps a function object with the methods of an `IDocumentable`, giving us the behavior we need to document it. Now all we need to do is define similar wrappers for all the other built-in types, and for any user-defined types, and then pick the right kind of wrapper to use when given an object to document.

But wait! Isn't this just as complicated as writing a documentation tool the "old fashioned" way? We still have to write code to get the data we need, *and* we still need to figure out what code to use. Where's the benefit?

Enter the PEP 246 `adapt()` function. `adapt()` has two required arguments: an object to adapt, and a **protocol** that you want it adapted to. Our documentation tool, when given an object to document, will simply call `adapt(object,IDocumentable)` and receive an instance of the appropriate adapter. (Or, if the object has already declared that it supports `IDocumentable`, we'll simply get the same object back from `adapt()` that we passed in.)

But that's only the beginning. If we create and distribute a documentation tool based on `IDocumentable`, then anyone who creates a new kind of documentable object can write their own adapters, and register them via the `protocols` package, *without* changing the documentation tool, or needing to give special configuration options to the tool or call tool-specific registration functions. (Which means we don't have to design or code those options or registration functions into our tool.)

Also, lets say I use some kind of "finite state machine" library written by vendor A, and I'd like to use it with this new documentation tool from vendor B. I can write and register adapters from such types as "finite state machine," "state," and "transition" to `IDocumentable`. I can then use vendor B's tool to document vendor A's object types.

And it goes further. Suppose vendor C comes out with a new super-duper documentation framework with advanced features. To use the new features, however, a more sophisticated interface than `IDocumentable` is needed. So vendor C's tool requires objects to support his new `ISuperDocumentable` interface. What happens then? Is the new package doomed to sit idle because everybody else only has `IDocumentable` objects?

Heck no. At least, not if vendor C starts by defining an adapter from `IDocumentable` to `ISuperDocumentable` that supplies reasonable default behavior for "older" objects. Then he or she writes adapters from built-in types to `ISuperDocumentable` that provide more useful-than-default behaviors where applicable. Now, the

super-framework is instantly usable with existing adapters for other object types. And, if vendor C defined `ISuperDocumentable` as *extending* `IDocumentable`, there's another benefit, too.

Suppose vendor A upgrades their finite state machine library to include direct or adapted support for the new `ISuperDocumentable` interface. Do I need to switch to vendor C's documentation tool now? Must I continue to maintain my FSM-to-`IDocumentable` adapters? No, to both questions. If `ISuperDocumentable` is a strict extension of `IDocumentable`, then I may use an `ISuperDocumentable` anywhere an `IDocumentable` is accepted. Thus, I can use vendor A's new library with my existing (non-"super") documentation tool from vendor B, without needing my old adapters any more.

As you can see, replacing introspection with adaptation makes frameworks more flexible and extensible. It also simplifies the design process, by letting you focus on what you *want* to do, instead of on the details of which objects are doing it.

As we mentioned at the beginning of this section, this example is only a sketch of a skeleton of a real documentation framework. For example, a real documentation framework would also need to define an `ISignature` interface for objects returned from `getSignature()`. We've also glossed over many other issues that the designers of a real documentation framework would face, in order to focus on the problems that can be readily solved with adaptable components.

And that's our point, actually. Every framework has two kinds of design issues: the ones that are specific to the framework's subject area, and the ones that would apply to any framework. The `protocols` package can save you a lot of work dealing with the latter, so you can spend more time focusing on the former. Let's start looking at how, beginning with the concepts of "protocols" and "interfaces".

1.1.2 Protocols and Interfaces

Many languages and systems provide ways of defining **interfaces** that components provide or require. Some mechanisms are purely for documentation, others are used at runtime to obtain or verify an implementation. Typically, interfaces are formal, intended for compiler-verified static type checking.

As a dynamic language, Python more often uses a looser notion of interface, known as a **protocol**. While protocols are often very precisely specified, their intended audience is a human reader or developer, not a compiler or automated verification tool.

Automated verification tools, however, usually extract a high overhead cost from developers. The Java language, for example, requires that all methods of an interface be defined by a class that claims to implement the interface, even if those methods are never used in the program being compiled! And yet, the more important *dynamic* behavior of the interface at runtime is not captured or verifiable by the compiler, so written documentation for human readers is still required!

In the Python language, the primary uses for objects representing protocols or interfaces are at runtime, rather than at compile time. Typically, such objects are used to ask for an implementation of the interface, or supplied by an object to claim that it provides an implementation of that interface.

In principle, any Python object may be used as a **protocol object**. However, for a variety of practical reasons, it is best that protocol objects be hashable and comparable. That is, protocol objects should be usable as dictionary keys.

This still allows for a wide variety of protocol object implementations, however. One might assign meaning to the number 42, for example, as referring to some hypothetical "hitchhiker" protocol. More realistically, the Microsoft COM framework uses UUIDs (Universally Unique Identifiers) to identify interfaces. UUIDs can be represented as Python strings, and thus are usable as protocol objects.

But a simple string or number is often not very useful as a protocol object. Aside from the issue of how to assign strings or numbers to protocols, these passive protocol objects cannot *do* anything, and by themselves they document nothing.

There are thus two more common approaches to creating protocol objects in Python: classes (such as abstract base classes or "ABCs"), and **interface objects**. Interface objects are typically also defined using Python `class` state-

ments, but use a custom metaclass to create an object that may not be usable in the same ways as a "real" Python class. Many Python frameworks (such as Twisted, Zope, and this package) provide their own framework-specific implementations of this "interface object" approach.

Since classes and most interface object implementations can be used as dictionary keys, and because their Python source code can serve as (or be converted to) useful documentation, both of these approaches are viable ways to create protocol objects usable with the `protocols` package.

In addition, inheriting from a class or interface objects is a simple way to define implication relationships between protocol objects. Inheriting from a protocol to create a new protocol means that the new protocol **implies** the old protocol. That is, any implementation or adaptation to the new protocol, is implied to be usable in a place where the old protocol was required. (We will have more to say about direct and adapted implication relationships later on, in section 1.1.7.)

At this point, we still haven't described any mechanisms for making adapters available, or declaring what protocols are supported by a class or object. To do that, we need to define two additional kinds of protocol objects, that have more specialized abilities.

An **adapting protocol** is a protocol object that is potentially able to adapt components to support the protocol it represents, or at least to recognize that a component supports (or claims to support) the protocol. To do this, an adapting protocol must have an `__adapt__` method, as will be described in section 1.1.3. (Often, this method can be added to an existing class, or patched into an interface object implementation.)

An **open protocol** is an adapting protocol that is also capable of accepting adapter declarations, and managing its implication relationships with other protocols. Open protocols can be used with this package's protocol declaration API, as long as they implement (or can be adapted to) the `IOpenProtocol` interface, as will be described in section 1.1.5.

Notice that the concepts of protocol objects, adapting protocols, and open protocols are themselves "protocols". The `protocols` package supplies three interface objects that symbolize these concepts: `IProtocol`, `IAdaptingProtocol`, and `IOpenProtocol`, respectively. Just as the English phrases represent the concepts in this text, the interface objects represent these concepts at runtime.

Whether a protocol object is as simple as a string, or as complex as an `IOpenProtocol`, it can be used to request that a component provide (or be adaptable to) the protocol that it symbolizes. In the next section, we'll look at how to make such a request, and how the different kinds of protocol objects participate (or not) in fulfilling such requests.

1.1.3 `adapt()` and the Adaptation Protocol

Component adaptation is the central focus of the `protocols` package. All of the package's protocol declaration API depends on component adaptation in order to function, and the rest of the package is just there to make it easier for developers to use component adaptation in their frameworks and programs.

Component adaptation is performed by calling the `adapt()` function, whose design is based largely on the specification presented in PEP 246:

`adapt`(*component, protocol,* [*, default*])
: Return an implementation of *protocol* (a protocol object) for *component* (any object). The implementation returned may be *component*, or a wrapper that implements the protocol on its behalf. If no implementation is available, return *default*. If no *default* is provided, raise `protocols.AdaptationFailure`.

The component adaptation process performed by `adapt()` proceeds in four steps:

1. If *protocol* is a class or type, and *component* is an instance of that class or type, the component is returned unchanged. (This quickly disposes of the most trivial cases).

2. If *component* has a `__conform__` method, it is called, passing in the protocol. If the method returns a value other than `None`, it is returned as the result of `adapt()`.

3. If *protocol* has an `__adapt__` method, it is called, passing in *component*. If the method returns a value other than `None`, it is returned as the result of `adapt()`.

4. Perform default processing as described above, returning *default* or raising `protocols.AdaptationFailure` as appropriate.

This four-step process is called the **adaptation protocol**. Note that it can be useful even in the case where neither the component nor the protocol object are aware that the adaptation protocol exists, and it gracefully degrades to a kind of `isinstance()` check in that case. However, if either the component or the protocol object has been constructed (or altered) so that it has the appropriate `__conform__` or `__adapt__` method, then much more meaningful results can be achieved.

Throughout the rest of this document, we will say that a component **supports** a protocol, if calling `adapt(component,protocol)` does not raise an error. That is, a component supports a protocol if its `__conform__` method or the protocol's `__adapt__` method return a non-`None` value.

This is different from saying that an object **provides** a protocol. An object provides a protocol if `adapt(ob,protocol) is ob`. Thus, if an object *provides* a protocol, it *supports* the protocol, but an object can also support a protocol by having an adapter that provides the protocol on its behalf.

Now that you know how `adapt()` works, you can actually make use of it without any of the other tools in the `protocols` package. Just define your own `__conform__` and `__adapt__` methods, and off you go!

In practice, however, this is like creating a new kind of Python "number" type. That is, it's certainly possible, but can be rather tedious and is perhaps best left to a specialist. For that reason, the `protocols` package supplies some useful basic protocol types, and a "declaration API" that lets you declare how protocols, types, and objects should be adapted to one another. The rest of this document deals with how to use those types and APIs.

You don't need to know about those types and APIs to create your own kinds of protocols or components, just as you don't need to have studied Python's numeric types or math libraries to create a numeric type of your own. But, if you'd like your new types to interoperate well with existing types, and conform to users' expectations of how such a type behaves, it would be a good idea to be familiar with existing implementations, such as the ones described here.

Creating and Using Adapters, Components, and Protocols

Because the adaptation protocol is so simple and flexible, there are a few guidelines you should follow when using `adapt()` or creating `__conform__` and `__adapt__` methods, to ensure that adapted objects are as usable as unadapted objects.

First, adaptation should be **idempotent**. That is, if you `adapt()` an object to a protocol, and then `adapt()` the return value to the same protocol, the same object should be returned the second time. If you are using the `protocols` declaration API, it suffices to declare that instances of the adapter class provide the protocol they adapt to. That is, if an adapter class provides protocol P for objects of type X, then it should declare that it provides protocol P.

If you are not using the declaration API, but relying only upon your custom `__conform__` and `__adapt__` methods, you need to ensure that any adapters you return will return themselves when asked to support the protocol that they were returned as an adapter for.

Second, adaptation is not automatically **symmetric**. That is, if I have an object X that provides protocol P1, and I `adapt()` it to protocol P2, it is not guaranteed that I can `adapt()` the resulting object to P1 and receive the original object. Ideally, someone who defines an adapter function would also declare an inverse adapter function to "unwrap" an adapted object to its original identity. In practice, however, this can be complex, since the adapter might need some fairly global knowledge of the system to know when it is better to unwrap and rewrap, and when it is better to further wrap the existing wrapper.

Another issue that occurs with such wrapper-based adaptation, is that the wrapper does not have the same object identity as the base object, and may not hash or compare equal to it, either. Further, it is not guaranteed that subsequent calls to `adapt()` will yield the same wrapper object – in fact it's quite unlikely.

These characteristics of adapted objects can be easily dealt with, however, by following a few simple rules:

- Always `adapt()` from the "original" object you're supplied; avoid adapting adaptations.
- Always pass "original" objects to functions or methods that expect their input to support more than one protocol; only pass adapted objects to functions or methods that expect support for only one protocol.
- Always use "original" objects for equality or identity comparisons – or else ensure that callers know they will need to provide you with an equal or identical adapter. (One good way to document this, is to include the requirement in the definition of the interface or protocol that your system requires.)

In some respects, these rules are similar to dealing with objects in statically typed languages like Java. In Java, if one simply has an "object", it is not possible to perform operations specific to an interface, without first "casting" the object to that interface. But, the object that was "cast" can't be stored in the same variable that the "object" was in, because it is of a different type.

Replacing Introspection with Adaptation

> To summarize: don't type check.
>
> — Alex Martelli, on comp.lang.python

Component adaptation is intended to completely replace all non-cooperative introspection techniques, such as `type()`, `isinstance()`, `hasattr()`, and even interface checks. Such introspection tends to limit framework flexibility by unnecessarily closing policies to extension by framework users. It often makes code maintenance more difficult as well, since such checks are often performed in more than one place, and must be kept in sync whenever a new interface or type must be checked.

Some common use cases for such introspection are:

- To manually adapt a supplied component to a needed interface
- To select one of several possible behaviors, based on the kind of component supplied
- To select another component, or take some action, using information about the interfaces supported by the supplied component

Obviously, the first case is handled quite well by `adapt()`, at least in an environment where it's easy to declare adapters between types and protocols. The second and third cases may at first seem to demand an ability to introspect what interfaces are supported by a component. But they are almost always better served by defining new protocols that supply the required behavior or metadata, and then requesting implementations of those protocols.

In all three use cases, replacing introspection with adaptation opens the framework to third party extensions, without further modifications being required – and without the need to do extensive design or documentation of a new hook or extension point to be added to the framework. Indeed, the availability of a standard mechanism for adaptation means that the extension mechanism need only be documented once: right here in this document.

In section 1.1.11, we will present techniques for refactoring all three kinds of introspection code to purely adaptation-driven code, showing how the flexibility and readability of the code improves in the process. But first, we will need to cover how protocols and interfaces can be defined, declared, and adapted, using the API provided by the `protocols` package.

See Also:

isinstance() Considered Harmful
(http://www.canonical.org/~kragen/isinstance/)
: A brief critique of common justifications for using introspection

Differences Between `protocols.adapt()` and PEP 246

If you have read PEP 246 or are looking for an exact implementation of it, you should know that there are a few differences between the `protocols` implementation of `adapt()` and the PEP 246 specification. If you don't care about these differences, you can skip this mini-appendix and proceed directly to section 1.1.4, "Defining and Subclassing Interfaces".

The first difference is that `TypeError` is treated differently in each implementation. PEP 246 says that if a `__conform__` or `__adapt__` method raises a `TypeError`, it should be treated in the same way as if the method returned `None`. This was a workaround for the issue of accidentally calling an unbound class method, in the case where a component or protocol supplied to `adapt()` was a class.

The `protocols` implementation of `adapt()` attempts to catch such errors also, but will reraise any exception that appears to come from *within* the execution of the `__conform__` or `__adapt__` method. So if these methods raise a `TypeError`, it will be passed through to the caller of `adapt`. Thus, if you are writing one of these methods, you should not raise a `TypeError` to signal the lack of an adaptation. Rather, you should return `None`.

Second, `protocols.AdaptationFailure` is raised when no adaptation is found, and no default is supplied, rather than the `TypeError` specified by PEP 246. (**Note:** `protocols.AdaptationFailure` is a subclass of `TypeError` and `NotImplementedError`, so code written to catch either of these errors will work.)

These differences are the result of experience using the `protocols` package with PEAK, and advances in the Python state-of-the-art since PEP 246 was written (over two years ago). We believe that they make the adaptation protocol more robust, more predictable, and easier to use for its most common applications.

Convenience Adaptation API (NEW in 0.9.3)

As of version 0.9.3, PyProtocols supports the simplified adaptation API that was pioneered by Twisted, and later adopted by Zope. In this simplified API, a protocol can be called, passing in the object to be adapted. So, for example, instead of calling `adapt(foo,IBar)`, one may call `IBar(foo)`. The optional *default* arguments can also be supplied, following the *component* parameter.

All of the protocol types supplied by PyProtocols now support this simpler calling scheme, except for `AbstractBase` subclasses, because calling an `AbstractBase` subclass should create an instance of that subclass, not attempt to adapt an arbitrary object.

Notice, by the way, that you should only use this simplified API if you know for certain that the protocol supports it. For example, it's safe to invoke a known, constant interface object in this way. But if you're writing code that may receive a protocol object as a parameter or via another object, you should use `adapt()` instead, because you may receive a protocol object that does not support this shortcut API.

1.1.4 Defining and Subclassing Interfaces

The easiest way to define an interface with the `protocols` package is to subclass `protocols.Interface`. `Interface` does not supply any data or methods of its own, so you are free to define whatever you need. There are two common styles of defining interfaces, illustrated below:

```
from protocols import Interface, AbstractBase

# "Pure" interface style

class IReadMapping(Interface):

    """A getitem-only mapping"""

    def __getitem__(key):
        """Return value for key"""


# Abstract Base Class (ABC) style

class AbstractMapping(AbstractBase):

    """A getitem-only mapping"""

    def __getitem__(self,key):
        """Return value for key"""
        raise NotImplementedError
```

The “pure” style emphasizes the interface as seen by the caller, and is not intended to be subclassed for implementation. Notice that the `self` parameter is not included in its method definitions, because `self` is not supplied when calling the methods. The “ABC” style, on the other hand, emphasizes implementation, as it is intended to be subclassed for that purpose. Therefore, it includes method bodies, even for abstract methods. Each style has different uses: “ABC” is a popular rapid development style, while the “pure” approach has some distinct documentation advantages.

`protocols.AbstractBase` may be used as a base class for either style, but `protocols.Interface` is only usable for the ”pure” interface style, as it supports the convenience adaptation API (see section 1.1.3).

(**Note:** both base classes use an explicit metaclass, so keep in mind that if you want to subclass an abstract base for implementation using a different metaclass, you may need to create a third metaclass that combines `protocols.AbstractBaseMeta` with your desired metaclass.)

Subclassing a subclass of `Interface` (or `AbstractBase`) creates a new interface (or ABC) that implies the first interface (or ABC). This means that any object that supports the second interface (or ABC), is considered to implicitly support the first interface (or ABC). For example:

```
class IReadWriteMapping(IReadMapping):

    """Mapping with getitem and setitem only"""

    def __setitem__(key,value):
        """Store value for key, return None"""
```

The `IReadWriteMapping` interface implies the `IReadMapping` interface. Therefore, any object that supports `IReadWriteMapping` is understood to also support the `IReadMapping` interface. The reverse, however, is not true.

Inheritance is only one way to declare that one interface implies another, however, and its uses are limited. Let's say for example, that some package `A` supplies objects that support `IReadWriteMapping`, while package `B` needs objects that support `IReadMapping`. But each package declared its own interface, neither inheriting from the other.

As developers reading the documentation of these interfaces, it is obvious to us that `IReadWriteMapping` implies `IReadMapping`, because we understand what they do. But there is no way for Python to know this, unless we explicitly state it, like this:

```
import protocols
from A import IReadWriteMapping
from B import IReadMapping

protocols.declareAdapter(
    protocols.NO_ADAPTER_NEEDED,
    provides = [IReadMapping],
    forProtocols = [IReadWriteMapping]
)
```

In the above example, we use the `protocols` declaration API to say that no adapter is needed to support the `B.IReadMapping` interface for objects that already support the `A.IReadWriteMapping` interface.

At this point, if we supply an object that supports `IReadWriteMapping`, to a function that expects an `IReadMapping`, it should work, as long as we call `adapt(ob,IReadMapping)` (or `IReadMapping(ob)`) first, or the code we're calling does so.

There are still other ways to declare that one interface implies another. For example, if the author of our example package `B` knew about package A and its `IReadWriteMapping` interface, he or she might have defined `IReadMapping` this way:

```
import protocols
from protocols import Interface

from A import IReadWriteMapping

class IReadMapping(Interface):

    """A getitem-only mapping"""

    protocols.advise(
        protocolIsSubsetOf = [IReadWriteMapping]
    )

    def __getitem__(key):
        """Return value for key"""
```

This is syntax sugar for creating the interface first, and then using `protocols.declareAdapter(NO_ADAPTER_NEEDED)`. Of course, you can only use this approach if you are the author of the interface! Otherwise, you must use `declareAdapter()` after the fact, as in the previous example.

In later sections, we will begin looking at the `protocols` declaration APIs – like `declareAdapter()` and `advise()` – in more detail. But first, we must look briefly at the interfaces that the `protocols` package expects from the protocols, adapters, and other objects supplied as parameters to the declaration API.

1.1.5 Interfaces Used by the Declaration API

Like any other API, the `protocols` declaration API has certain expectations regarding its parameters. These expectations are documented and referenced in code using interfaces defined in the `protocols.interfaces` module. (The interfaces are also exported directly from the top level of the `protocols` package.)

You will rarely use or subclass any of these interface objects, unless you are customizing or extending the system. Four of the interfaces exist exclusively for documentation purposes, while the rest are used in `adapt()` calls made by the API.

First, let's look at the documentation-only interfaces. It is not necessary for you to declare that an object supports these interfaces, and the `protocols` package never tries to `adapt()` objects to them.

IAdapterFactory
: Up until this point, we've been talking about "adapters" rather loosely. The `IAdapterFactory` interface formalizes the concept. An **adapter factory** is a callable object that accepts an object to be adapted, and returns an object that provides the protocol on behalf of the passed-in object. Declaration API functions that take "adapter" or "factory" arguments must be adapter factories.

  The `protocols` package supplies two functions that provide this interface: `NO_ADAPTER_NEEDED` and `DOES_NOT_SUPPORT`. `NO_ADAPTER_NEEDED` is used to declare that an object provides a protocol directly, and thus it returns the object passed into it, rather than some kind of adapter. `DOES_NOT_SUPPORT` is used to declare that an object does not support a protocol, even with an adapter. (Since this is the default case, `DOES_NOT_SUPPORT` is rarely used, except to indicate that a subclass does not support an interface that one of its superclasses does.)

IProtocol
: This interface formalizes the idea of a "protocol object". As you will recall from section 1.1.2, a protocol object is any object that can be used as a Python dictionary key. The second argument to `adapt()` must be a protocol object according to this definition.

IAdaptingProtocol
: This interface formalizes the idea of an "adapting protocol", specifically that it is a protocol object (i.e., it provides `IProtocol`) that also has an `__adapt__` method as described in section 1.1.3. `IAdaptingProtocol` is a subclass of `IProtocol`, so of course `adapt()` accepts such objects as protocols.

IImplicationListener
: This interface is for objects that want to receive notification when new implication relationships (i.e. adapters) are registered between two protocols. If you have objects that want to keep track of what interfaces they support, you may want those object to implement this interface so they can be kept informed of new protocol-to-protocol adapters.

The other three interfaces are critical to the operation of the declaration API, and thus must be supported by objects supplied to it. The `protocols` package supplies and registers various adapter classes that provide these interfaces on behalf of many commonly used Python object types. So, for each interface, we will list "known supporters" of that interface, whether they are classes supplied by `protocols`, or built-in types that are automatically adapted to the interface.

We will not, however, go into details here about the methods and behavior required by each interface. (Those details can be found in section 1.1.9.)

IOpenProtocol
: This interface formalizes the "open protocol" concept that was introduced in section 1.1.2. An `IOpenProtocol` is an `IAdaptingProtocol` that can also accept declarations made by the `protocols` declaration API.

The `protocols` package supplies two implementations of this interface: `Protocol` and `InterfaceClass`. Thus, any `Interface` subclass or `Protocol` instance is automatically considered to provide `IOpenProtocol`. **Note:** `Interface` is an instance of `InterfaceClass`, and thus provides `IOpenProtocol`. But if you create an instance of an `Interface`, that object does not provide `IOpenProtocol`, because the interfaces provided by an object and its class (or its instances) can be different.

In addition to its built-in implementations, the `protocols` package also supplies and can declare adapter factories that adapt Zope X3 and Twisted's interface objects to the `IOpenProtocol` interface, thus allowing you to use Zope and Twisted interfaces in calls to the declaration API. Similar adapters for other frameworks' interfaces may be added, if there is sufficient demand and/or contributed code, and the frameworks' authors do not add the adapters to their frameworks.

IOpenImplementor

An `IOpenImplementor` is a class or type that can be told (via the declaration API) what protocols its instances provide (or support via an `IAdapterFactory`). Note that this implies that the instances have a `__conform__` method, or else they would not be able to tell `adapt()` about the declared support!

Support for this interface is optional, since types that don't support it can still have their instances be adapted by `IOpenProtocol` objects. The `protocols` package does not supply any implementations or adapters for this interface, either. It is intended primarily as a hook for classes to be able to receive notification about protocol declarations for their instances.

IOpenProvider

Because objects' behavior usually comes from a class definition, it's not that often that you will declare that a specific object provides or supports an interface. But objects like functions and modules do not have a class definition, and classes themselves sometimes provide an interface. (For example, one could say that class objects provide an `IClass` interface.) So, the declaration API needs to also be able to declare what protocols an individual object (such as a function, module, or class) supports or provides.

That's what the `IOpenProvider` interface is for. An `IOpenProvider` is an object with a `__conform__` method, that can be told (via the declaration API) what protocols it provides (or supports via an `IAdapterFactory`).

Notice that this is different from `IOpenImplementor`, which deals with an class or type's instances. `IOpenProvider` deals with the object itself. A single object can potentially be both an `IOpenProvider` and an `IOpenImplementor`, if it is a class or type.

The `protocols` package supplies and declares an adapter factory that can adapt most Python objects to support this interface, assuming that they have a `__dict__` attribute. Thus, it is acceptable to pass a Python function, module, or instance of a "classic" class to any declaration API that expects an `IOpenProvider` argument.

We'll talk more about making protocol declarations for individual objects (as opposed to types) in section 1.1.6, "Protocol Declarations for Individual Objects".

1.1.6 Declaring Implementations and Adapters

There are three kinds of relationships that a protocol can participate in:

- A relationship between a class or type, and a protocol its instances provide or can be adapted to,
- A relationship between an instance, and a protocol it provides or can be adapted to, and
- A relationship between a protocol, and another protocol that it implies or can be adapted to.

Each of these relationships is defined by a **source** (a type, instance or protocol), a **destination** (desired) protocol, and an **adapter factory** used to convert from one to the other. If no adapter is needed, we can say that the adapter factory is the special `NO_ADAPTER_NEEDED` function.

To declare relationships like these, the `protocols` declaration API provides three "primitive" declaration functions. Each accepts a destination protocol (that must support the `IOpenProtocol` interface), an adapter factory (or `NO_ADAPTER_NEEDED`), and a source (type, instance, or protocol). These three functions are `declareAdapterForType()`, `declareAdapterForObject()`, and `declareAdapterForProtocol()`, respectively.

You will not ordinarily use these primitives, however, unless you are customizing or extending the framework. It is generally easier to call one of the higher level functions in the declaration API. These higher-level functions may make several calls to the primitive functions on your behalf, or supply useful defaults for certain parameters. They are, however, based entirely on the primitive functions, which is important for customizations and extensions.

The next higher layer of declaration APIs are the explicit declaration functions: `declareImplementation`, `declareAdapter`, and `adviseObject`. These functions are structured to support the most common declaration use cases.

For declaring protocols related to a type or class:

`declareImplementation`(*typ* [, *instancesProvide=[ ]*] [, *instancesDoNotProvide=[ ]*])

Declare that instances of class or type *typ* do or do not provide implementations of the specified protocols. *instancesProvide* and *instancesDoNotProvide* must be sequences of protocol objects that provide (or are adaptable to) the `IOpenProtocol` interface, such as `protocols.Interface` subclasses, or `Interface` objects from Zope or Twisted.

This function is shorthand for calling `declareAdapterForType()` with `NO_ADAPTER_NEEDED` and `DOES_NOT_SUPPORT` as adapters from the type to each of the specified protocols. Note, therefore, that the listed protocols must be adaptable to `IOpenProtocol`. See `declareAdapterForType()` in section 1.1.9 for details.

For declaring protocols related to a specific, individual instance:

`adviseObject`(*ob* [, *provides=[ ]*] [, *doesNotProvide=[ ]*])

Declare that *ob* provides (or does not provide) the specified protocols. This is shorthand for calling `declareAdapterForObject()` with `NO_ADAPTER_NEEDED` and `DOES_NOT_SUPPORT` as adapters from the object to each of the specified protocols. Note, therefore, that *ob* may need to support `IOpenProvider`, and the listed protocols must be adaptable to `IOpenProtocol`. See `declareAdapterForObject()` in section 1.1.9 for details. Also, see section 1.1.6, "Protocol Declarations for Individual Objects", for more information on using `adviseObject`.

And for declaring all other kinds of protocol relationships:

`declareAdapter`(*factory, provides,* [, *forTypes=[ ]*] [, *forProtocols=[ ]*] [, *forObjects=[ ]*])

Declare that *factory* is an `IAdapterFactory` whose return value provides the protocols listed in *provides* as an adapter for the classes/types listed in *forTypes*, for objects providing the protocols listed in *forProtocols*, and for the specific objects listed in *forObjects*.

This function is shorthand for calling the primitive declaration functions for each of the protocols listed in *provides* and each of the sources listed in the respective keyword arguments.

Although these forms are easier to use than raw `declareAdapterForX` calls, they still require explicit reference to the types or objects involved. For the most common use cases, such as declaring protocol relationships to a class, or declaring an adapter class, it is usually easier to use the "magic" `protocols.advise()` function, which we will discuss next.

Convenience Declarations in Class, Interface and Module Bodies

Adapters, interfaces, and protocol implementations are usually defined in Python `class` statements. To make it more convenient to make protocol declarations for these classes, the `protocols` package supplies the `advise()` function. This function can make declarations about a class, simply by being called from the body of that class. It can also be called from the body of a module, to make a declaration about the module.

`advise`(***kw*)

Declare protocol relationships for the containing class or module. All parameters must be supplied as keyword arguments. This function must be called directly from a class or module body, or a `SyntaxError` results at runtime. Different arguments are accepted, according to whether the function is called within a class or module.

When invoked in the top-level code of a module, this function only accepts the `moduleProvides` keyword argument. When invoked in the body of a class definition, this function accepts any keyword arguments *except* `moduleProvides`. The complete list of keyword arguments follows. Unless otherwise specified, protocols must support the `IOpenProtocol` interface.

Note: When used in a class body, this function works by temporarily replacing the `__metaclass__` of the class. If your class sets an explicit `__metaclass__`, it must do so *before* `advise()` is called, or the protocol declarations will not occur!

Keyword arguments accepted by `advise()`:

instancesProvide = *protocols*

A sequence of protocols that instances of the containing class provide, without needing an adapter. Supplying this argument is equivalent to calling `declareImplementation(`*containing class*`,`*protocols*`)`.

instancesDoNotProvide = *protocols*

A sequence of protocols that instances of the containing class do not provide. This is primarily intended for "rejecting" protocols provided or supported by base classes of the containing class. Supplying this argument is equivalent to calling `declareImplementation(`*containing class*`,instancesDoNotProvide=`*protocols*`)`.

asAdapterForTypes = *types*

Declare the containing class as an adapter for *types*, to the protocols listed by the `instancesProvide` argument (which must also be supplied). Supplying this argument is equivalent to calling `declareAdapter(`*containing class*`, `*instancesProvide*`, forTypes=`*types*`)`. (Note that this means the containing class must be an object that provides `IAdapterFactory`; i.e., its constructor should accept being called with a single argument: the object to be adapted.)

asAdapterForProtocols = *protocols*

Declare the containing class as an adapter for *protocols*, to the protocols listed by the `instancesProvide` argument (which must also be supplied). Supplying this argument is equivalent to calling `declareAdapter(`*containing class*`, `*instancesProvide*`, forProtocols=`*types*`)`. (Note that this means the containing class must be an object that provides `IAdapterFactory`; i.e., its constructor should accept being called with a single argument: the object to be adapted.)

factoryMethod = *methodName*

New in version 0.9.1. When using `asAdapterForTypes` or `asAdapterForProtocols`, you can also supply a factory method name, using this keyword. The method named must be a *class* method, and it will be used in place of the class' normal constructor. (Note that this means the named method must be able to be called with a single argument: the object to be adapted.)

protocolExtends = ***protocols***

Declare that the containing class is a protocol that extends (i.e., implies) the listed protocols. This keyword argument is intended for use inside class statements that themselves define protocols, such as `Interface` subclasses, and that need to "inherit" from incompatible protocols. For example, an `Interface` cannot directly subclass a Zope interface, because their metaclasses are incompatible. But using `protocolExtends` works around this:

```
import protocols
from mypackage import ISomeInterface
from zope.something.interfaces import ISomeZopeInterface

class IAnotherInterface(ISomeInterface):
    protocols.advise(
        protocolExtends = [ISomeZopeInterface]
    )
    #... etc.
```

In the above example, `IAnotherInterface` wants to extend both `ISomeInterface` and `ISomeZopeInterface`, but cannot do so directly because the interfaces are of incompatible types. `protocolExtends` informs the newly created interface that it implies `ISomeZopeInterface`, even though it isn't derived from it.

Using this keyword argument is equivalent to calling `declareAdapter(NO_ADAPTER_NEEDED,` *protocols*`, forProtocols=[`*containing class*`])`. Note that this means that the containing class must be an object that supports `IOpenProtocol`, such as an `Interface` subclass.

protocolIsSubsetOf = ***protocols***

Declare that the containing class is a protocol that is implied (extended) by the listed protocols. This is just like `protocolExtends`, but in the "opposite direction". It allows you to declare (in effect) that some other interface is actually a subclass of (extends, implies) this one. See the examples in section 1.1.4 for illustration.

Using this keyword argument is equivalent to calling `declareAdapter(NO_ADAPTER_NEEDED, [`*containing class*`], forProtocols=`*protocols*`)`.

equivalentProtocols = ***protocols***

New in version 0.9.1. Declare that the containing class is a protocol that is equivalent to the listed protocols. That is, the containing protocol both implies, and is implied by, the listed protocols. This is a convenience feature intended mainly to support the use of generated protocols.

Using this keyword is equivalent to using both the `protocolExtends` and `protocolIsSubsetOf` keywords, with the supplied protocols.

classProvides = ***protocols***

Declare that the containing class *itself* provides the specified protocols. Supplying this argument is equivalent to calling `adviseObject(`*containing class*`,` *protocols*`)`. Note that this means that the containing class may need to support the `IOpenProvider` interface. The `protocols` package supplies default adapters to support `IOpenProvider` for both classic and new-style classes, as long as they do not have custom `__conform__` methods. See section 1.1.6, "Protocol Declarations for Individual Objects" for more details.

classDoesNotProvide = ***protocols***

Declare that the containing class *itself* does not provide the specified protocols. This is for classes that need to reject inherited class-level `classProvides` declarations. Supplying this argument is equivalent to calling `adviseObject(`*containing class*`, doesNotProvide=`*protocols*`)`, and the `IOpenProvider` requirements mentioned above for `classProvides` apply here as well.

moduleProvides = ***protocols*** (module context only)

A sequence of protocols that the enclosing module provides. Equivalent to `adviseObject(`*containing module*`,` *protocols*`)`.

Protocol Declarations for Individual Objects

Because objects' behavior usually comes from a class definition, it's not too often that you will declare that an individual object provides or supports an interface, as opposed to making a blanket declaration about an entire class or type of object. But objects like functions and modules do not *have* a class definition that encompasses their behavior, and classes themselves sometimes provide an interface (e.g. via `classmethod` objects).

So, the declaration API needs to also be able to declare what protocols an individual object (such as a function, module, or class) supports or provides. This is what `adviseObject()` and the `classProvides`/`classDoesNotProvide` keywords of `advise()` do.

In most cases, for an object to be usable with `adviseObject()`, it must support the `IOpenProvider` interface. Since many of the objects one might wish to use with `adviseObject()` (such as modules, functions, and classes) do not directly provide this interface, the `protocols.classic` module supplies and declares an adapter factory tha can adapt most Python objects to support this interface, assuming that they have a `__dict__` attribute.

This default adapter works well for many situations, but it has some limitations you may need to be aware of. First, it works by "poking" a new `__conform__` method into the adapted object. If the object already has a `__conform__` method, a `TypeError` will be raised. So, if you need an object to be an `IOpenProvider`, but it has a `__conform__` method, you may want to have its class include `ProviderMixin` among its base classes, so that your objects won't rely on the default adapter for `IOpenProvider`. (See `ProviderMixin` in section 1.1.9 for more on this.)

Both the default adapter and `ProviderMixin` support inheritance of protocol declarations, when the object being adapted is a class or type. In this way, `advise(classProvides=`*protocols*`)` declarations (or `adviseObject(someClass,`*protocols*`)` calls) are inherited by subclasses. Of course, you can always reject inherited protocol information using `advise(classDoesNotProvide=`*protocols*`)` or `adviseObject(newClass,doesNotProvide=`*protocols*`)`.

Both the default adapter and `ProviderMixin` work by keeping a mapping of protocols to adapter factories. Keep in mind that this means the protocols and adapter factories will continue to live until your object is garbage collected. Also, that means for your object to be pickleable, all of the protocols and adapter factories used must be pickleable. (This latter requirement can be quite difficult to meet, since composed adapter factories are dynamically created functions at present.)

Note that none of these restrictions apply if you are only using declarations about types and protocols, as opposed to individual objects. (Or if you only make individual-object declarations for functions, modules, and classes.) Also note that if you have some objects that need to dynamically support or not support a protocol on a per-instance basis, then `adviseObject()` is probably not what you want anyway! Instead, give your objects' class a `__conform__()` method that does the right thing when the object is asked to conform to a protocol. `adviseObject()` is really intended for adding metadata to objects that "don't know any better".

In general, protocol declarations are a *static* mechanism: they cannot be changed or removed at will, only successively refined. All protocol declarations made must be consistent with the declarations that have already been made. This makes them unsuitable as a mechanism for dynamic behavior such as supporting a protocol based on an object's current state.

In the next section, we'll look more at the static nature of declarations, and explore what it means to make conflicting (or refining) protocol declarations.

1.1.7 Protocol Implication and Adapter Precedence

So far, we've only dealt with simple one-to-one relationships between protocols, types, and adapter factories. We haven't looked, for example, at what happens when you define that class X instances provide interface IX, that AXY is an adapter factory that adapts interface IX to interface IY, and class Z subclasses class X. (As you might expect, what happens is that Z instances will be wrapped with an AXY adapter when you call `adapt(instanceOfZ, IY)`.)

Adaptation relationships declared via the declaration API are **transitive**. This means that if you declare an adaptation from item A to item B, and from item B to item C, then there is an **adapter path** from A to C. An adapter path is effectively a sequence of adapter factories that can be applied one by one to get from a source (type, object, or protocol) to a desired destination protocol.

Adapter paths are automatically composed by the types, objects, and protocols used with the declaration API, using the `composeAdapters()` function. Adapter paths are said to have a **depth**, which is the number of steps taken to get from the source to the destination protocol. For example, if factory AB adapts from A to B, and factory BC adapts from B to C, then an adapter factory composed of AB and BC would have a depth of 2. However, if we registered another adapter, AC, that adapts directly from A to C, this adapter path would have a depth of 1.

Naturally, adapter paths with lesser depth are more desirable, as they are less likely to be a "lossy" conversion, and are more likely to be efficient. For this reason, shorter paths take precedence over longer paths. Whenever an adapter factory is declared between two points that previously required a longer path, all adapter paths that previously included the longer path segment are updated to use the newly shortened route. Whenever an adapter factory is declared that would *lengthen* an existing path, it is ignored.

The net result is that the overall network of adapter paths will tend to stabilize over time. As an added benefit, it is safe to define circular adapter paths (e.g. A to B, B to C, C to A), as only the shortest useful adapter paths are generated.

We've previously mentioned the special adapter factories `NO_ADAPTER_NEEDED` and `DOES_NOT_SUPPORT`. There are a couple of special rules regarding these adapters that we need to add. Any adapter path that contains `DOES_NOT_SUPPORT` can be reduced to a single instance of `DOES_NOT_SUPPORT`, and any adapter path that contains `NO_ADAPTER_NEEDED` is equivalent to the same adapter path without it. These changes can be used to simplify adapter paths, but are only taken into consideration when comparing paths, if the "unsimplified" version of the adapter paths are the same length.

Lets' consider two adapter paths between A and C. Each proceeds by way of B. (i.e., they go from A to B to C.) Which one is preferable? Both adapters have a depth of 2, because there are two steps (A to B, B to C). But suppose one adapter path contains two arbitrary adapter factories, and the other is composed of one factory plus `NO_ADAPTER_NEEDED`. Clearly, that path is superior, since it effectively contains only one adapter instead of two.

This simplification, however, can *only* be applied when the unsimplified paths are of the same length. Why? Consider our example of two paths from A to B to C. If someone declares a direct path from A to C (i.e. not via B or any other intermediate protocol), we want this path to take precedence over an indirect path, even if both paths "simplify" to the same length. Only if we are choosing between two paths with the same number of steps can we can use the length of their simplified forms as a "tiebreaker".

So what happens when choosing between paths of the same number of steps and the same simplified length? A `TypeError` occurs, unless one of these conditions applies:

- One of the paths simplifies to `DOES_NOT_SUPPORT`, in which case the other path is considered preferable. (Some ability is better than none.)
- One of the paths simplifies to `NO_ADAPTER_NEEDED`, in which case it is considered preferable. (It's better not to have to adapt.)
- Both of the paths are the same object, in which case no change is required to the existing path. (The declaration is redundant.)

Notice that this means that it is not possible to override an existing adapter path unless you are improving on it a way visible to the system. This doesn't mean, however, that you can't take advantage of existing declarations, while still

overriding some of them.

Suppose that there exists a set of existing adapters and protocols defined by some frameworks, and we are writing an application using them. We would like, however, for our application to be able to override certain existing relationships. Say for example that we'd like to have an adapter path from A to C that's custom for our application, but we'd like to "inherit" all the other adaptations to C, so that by default any C implementation is still useful for our application.

The simple solution is to define a new protocol D as a **subset** of protocol C. This is effectively saying that `NO_ADAPTER_NEEDED` can adapt from C to D. All existing declarations adapting to C, are now usable as adaptations to D, but they will have lower precedence than any direct adaptation to D. So now we define our direct adaptation from A to D, and it will take precedence over any A to C to D path. But, any existing path that goes to C will be "inherited" by D.

Speaking of inheritance, please note that inheritance between types/classes has no effect on adapter path depth calculations. Instead, any path defined for a subclass takes absolute precedence over paths defined for a superclass, because the subclass is effectively a different starting point. In other words, if A is a class, and Q subclasses A, then an adapter path between Q and some protocol is a different path than the path between A and that protocol. There is no comparison between the two, and no conflict. However, if a path from Q to a desired protocol does not exist, then the existing best path for A will be used.

Sometimes, one wishes to subclass a class without taking on its full responsibilities. It may be that we want Q to use A's implementation, but we do not want to support some of A's protocols. In that case, we can declare `DOES_NOT_SUPPORT` adapters for those protocols, and these will ensure that the corresponding adapter paths for A are not used.

This is called **rejecting inherited declarations**. It is not, generally speaking, a good idea. If you want to use an existing class' implementation, but do not wish to abide by its contracts (protocols), you should be using **delegation** rather than inheritance. That is, you should define your new class so that it has an attribute that is an instance of the old class. For example, if you are tempted to subclass Python's built-in dictionary type, but you do not want your subclass to really *be* a dictionary, you should simply have an attribute that is a dictionary.

Because rejecting inherited declarations is a good indication that inheritance is being used improperly, the `protocols` package does not encourage the practice. Declaring a protocol as `DOES_NOT_SUPPORT` does not propagate to implied protocols, so every rejected protocol *must* be listed explicitly. If class A provided protocol B, and protocol B derived from (i.e. implied) protocol C, then you must explicitly reject both B and C if you do not want your subclass to support them.

See Also:

The logic of composing and comparing adapter paths is implemented via the `composeAdapters()` and `minimumAdapter()` functions in the `protocols.adapters` module. See section 1.1.9 for more details on these and other functions that relate to adapter paths.

1.1.8 Dynamic Protocols (NEW in 0.9.1)

For many common uses of protocols, it may be inconvenient to subclass `protocols.Interface` or to manually create a `Protocol` instance. So, the `protocols` package includes a number of utility functions to make these uses more convenient.

Defining a protocol based on a URI or UUID

`protocolForURI`(*uri*)

New in version 0.9.1. Return a protocol object that represents the supplied URI or UUID string. It is guaranteed that you will receive the same protocol object if you call this routine more than once with equal strings. This behavior is preserved even across pickling and unpickling of the returned protocol object.

The purpose of this function is to permit modules to refer to protocols defined in another module, that may or may not be present at runtime. To do this, a protocol author can declare that their protocol is equivalent to a URI string:

```
from protocols import advise, Interface, protocolForURI

class ISomething(Interface):
    advise(
        equivalentProtocols = [protocolForURI("some URI string")]
    )
    # etc...
```

Then, if someone wishes to use this protocol without importing `ISomething` (and thereby becoming dependent on the module that provides it), they can do something like:

```
from protocols import advise, protocolForURI

class MyClass:
    advise(
        provides = [protocolForURI("some URI string")]
    )
    # etc...
```

Thus, instances of `MyClass` will be considered to support `ISomething`, if needed. But, if `ISomething` doesn't exist, no error occurs.

Defining a protocol as a subset of an existing type

`protocolForType`(*baseType*, [*methods=(), implicit=False*])

Return a protocol object that represents the subset of *baseType* denoted by *methods*. It is guaranteed that you will receive the same protocol object if you call this routine more than once with eqivalent paremeters. This behavior is preserved even across pickling and unpickling of the returned protocol object.

baseType should be a type object, and *methods* should be a sequence of attribute or method names. (The order of the names is not important.) The *implicit* flag allows adapting objects that don't explicitly declare support for the protocol. (More on this later.)

Typical usage of this function is to quickly define a simple protocol based on a Python built-in type such as `list`, `dict`, or `file`:

```
IReadFile = protocols.protocolForType(file, ['read','close'])
IReadMapping = protocols.protocolForType(dict, ['__getitem__'])
```

The advantage of using this function instead of creating an `Interface` subclass is that users do not need to import your specific `Interface` definition. As long as they declare support for a proto-

col based on the same type, and with at least the required methods, then their object will be considered to support the protocol. For example, declaring that you support `protocolForType(file, ['read','write','close'])` automatically implies that you support `protocolForType(file, ['read','close'])` and `protocolForType(file, ['write','close'])` as well. (Note: instances of the *baseType* and its subclasses will also be considered to provide the returned protocol, whether or not they explicitly declare support.)

If you supply a true value for the *implicit* flag, the returned protocol will also adapt objects that have the specified methods or attributes. In other words, `protocolForType(file, ['read','close'], True)` returns a protocol that will consider any object with `read` and `close` methods to provide that protocol, as well as objects that explicitly support `protocolForType(file, ['read','close'])`.

In order to automatically declare the relationships between the protocols for different subsets, this function internally generates all possible subsets of a requested *methods* list. So, for example, requesting a protocol with 8 method names may cause as many as 127 protocol objects to be created. Of course, these are generated only once in the lifetime of the program, but you should be aware of this if you are using large method subsets. Using as few as 32 method names would create 2 billion protocols!

Note also that the supplied *baseType* is used only as a basis for semantic distinctions between sets of similar method names, and to declare that the *baseType* and its subclasses support the returned protocol. No protocol-to-protocol relationships are automatically defined between protocols requested for different base types, regardless of any subclass/superclass relationship between the base types.

Defining a protocol for a sequence

`sequenceOf`(*protocol*)

New in version 0.9.1. Return a protocol object that represents a sequence of objects adapted to *protocol*. Thus, `protocols.sequenceOf(IFoo)` is a protocol that represents a `protocols.IBasicSequence` of objects supporting the `IFoo` protocol. It is guaranteed that you will receive the same protocol object if you call this routine more than once with the same protocol, even across pickling and unpickling of the returned protocol object.

When this function creates a new sequence protocol, it automatically declares an adapter function from `protocols.IBasicSequence` to the new protocol. The adapter function returns the equivalent of `[adapt(x,protocol) for x in sequence]`, unless one of the adaptations fails, in which case it returns `None`, causing the adaptation to fail.

The built-in `list` and `tuple` types are declared as implementations of `protocols.IBasicSequence`, so protocols returned by `sequenceOf()` can be used immediately to convert lists or tuples into lists of objects supporting *protocol*. If you need to adapt other kinds of sequences using your `sequenceProtocol()`, you will need to declare that those sequences implement `protocols.IBasicSequence` unless they subclass `tuple`, `list`, or some other type that implements `protocols.IBasicSequence`.

Defining a protocol as a local variation of another protocol

class `Variation`(*baseProtocol* [, *context=None*])

New in version 0.9.1. A `Variation` is a `Protocol` that "inherits" adapter declarations from an existing protocol. When you create a `Variation`, it declares that it is implied by its *baseProtocol*, and so any adpater suitable for adapting to the base protocol is therefore suitable for the `Variation`. This allows you to then declare adapters to the variation protocol, without affecting those declared for the base protocol. In this way, you can have a protocol object that represents the use of the base protocol in a particular context. You can optionally specify that context via the *context* argument, which will then serve as the `context` attribute of the protocol. For more background on how this works and what it might be used for, see section 1.1.10.

1.1.9 Package Contents and Contained Modules

The following functions, classes, and interfaces are available from the top-level `protocols` package.

`adapt`(*component, protocol* [, *default*])

Return an implementation of *protocol* (a protocol object) for *component* (any object). The implementation returned may be *component*, or an adapter that implements the protocol on its behalf. If no implementation is available, return *default*. If no *default* is provided, raise `protocols.AdaptationFailure`.

A detailed description of this function's operations and purpose may be found in section 1.1.3.

exception `AdaptationFailure`

New in version 0.9.3. A subclass of `TypeError` and `NotImplementedError`, this exception type is raised by `adapt()` when no implementation can be found, and no *default* was supplied.

class `Adapter`(*ob, proto*)

New in version 0.9.1. This base class provides a convenient `__init__` method for adapter classes. To use it, just subclass `protocols.Adapter` and add methods to implement the desired interface(s). (And of course, declare what interfaces the adapter provides, for what types, and so on.) Your subclass' methods can use the following attribute, which will have been set by the `__init__` method:

`subject`

The `subject` attribute of an `Adapter` instance is the *ob* supplied to its constructor. That is, it is the object being adapted.

`advise`(***kw*)

Declare protocol relationships for the containing class or module. All parameters must be supplied as keyword arguments. This function must be called directly from a class or module body, or a `SyntaxError` results at runtime. Different arguments are accepted, according to whether the function is called within a class or module.

When invoked in the top-level code of a module, this function only accepts the `moduleProvides` keyword argument. When invoked in the body of a class definition, this function accepts any keyword arguments *except* `moduleProvides`. The complete list of keyword arguments can be found in section 1.1.6.

Note: When used in a class body, this function works by temporarily replacing the `__metaclass__` of the class. If your class sets an explicit `__metaclass__`, it must do so *before* `advise()` is called, or the protocol declarations will not occur!

`adviseObject`(*ob* [, *provides=[ ]*] [, *doesNotProvide=[ ]*])

Declare that *ob* provides (or does not provide) the specified protocols. This is shorthand for calling `declareAdapterForObject()` with `NO_ADAPTER_NEEDED` and `DOES_NOT_SUPPORT` as adapters from the object to each of the specified protocols. Note, therefore, that *ob* may need to support `IOpenProvider`, and the listed protocols must be adaptable to `IOpenProtocol`. See section 1.1.6, "Protocol Declarations for Individual Objects", for more information on using `adviseObject`.

class `Attribute`(*doc*[,*name=None, value=None*])

This class is used to document attributes required by an interface. An example usage:

```
from protocols import Interface, Attribute

class IFoo(Interface):

    Bar = Attribute("""All IFoos must have a Bar attribute""")
```

If you are using the "Abstract Base Class" or ABC style of interface documentation, you may wish to also use the *name* and *value* attributes. If supplied, the `Attribute` object will act as a data descriptor, supplying *value* as a default value, and storing any newly set value in the object's instance dictionary. This is useful if you will be subclassing the abstract base and creating instances of it, but still want to have documentation appear in the interface. When the interface is displayed with tools like **pydoc** or `help()`, the attribute documentation will be shown.

declareAdapter(*factory, provides* [, *forTypes=[ ]*] [, *forProtocols=[ ]*] [, *forObjects=[ ]*])

Declare that *factory* is an `IAdapterFactory` whose return value provides the protocols listed in *provides* as an adapter for the classes/types listed in *forTypes*, for objects providing the protocols listed in *forProtocols*, and for the specific objects listed in *forObjects*.

This function is shorthand for calling the primitive declaration functions (`declareAdapterForType`, `declareAdapterForProtocol`, and `declareAdapterForObject`) for each of the protocols listed in *provides* and each of the items listed in the respective keyword arguments.

declareImplementation(*typ* [, *instancesProvide=[ ]*] [, *instancesDoNotProvide=[ ]*])

Declare that instances of class or type *typ* do or do not provide implementations of the specified protocols. *instancesProvide* and *instancesDoNotProvide* must be sequences of protocol objects that provide (or are adaptable to) the `IOpenProtocol` interface, such as `protocols.Interface` subclasses, or `Interface` objects from Zope or Twisted.

This function is shorthand for calling `declareAdapterForType()` with `NO_ADAPTER_NEEDED` and `DOES_NOT_SUPPORT` as adapters from the type to each of the specified protocols. Note, therefore, that the listed protocols must be adaptable to `IOpenProtocol`.

DOES_NOT_SUPPORT(*component, protocol*)

This function simply returns `None`. It is a placeholder used whenever an object, type, or protocol does not implement or imply another protocol. Whenever adaptation is not possible, but the `protocols` API function you are calling requires an adapter, you should supply this function as the adapter. Some protocol implementations, such as the one for Zope interfaces, are unable to handle adapters other than `NO_ADAPTER_NEEDED` and `DOES_NOT_SUPPORT`.

class Interface

Subclass this to create a "pure" interface. See section 1.1.4 for more details.

class AbstractBase

New in version 0.9.3. Subclass this to create an "abstract base class" or "ABC" interface. See section 1.1.4 for more details.

NO_ADAPTER_NEEDED(*component, protocol*)

This function simply returns *component*. It is a placeholder used whenever an object, type, or protocol directly implements or implies another protocol. Whenever an adapter is not required, but the `protocols` API function you are calling requires an adapter, you should supply this function as the adapter. Some protocol implementations may be unable to handle adapters other than `NO_ADAPTER_NEEDED` and `DOES_NOT_SUPPORT`.

protocolForType(*baseType,* [*methods=(), implicit=False*])

New in version 0.9.1. Return a protocol object that represents the subset of *baseType* denoted by *methods*. It is guaranteed that you will receive the same protocol object if you call this routine more than once with eqivalent paremeters. This behavior is preserved even across pickling and unpickling of the returned protocol object.

baseType should be a type object, and *methods* should be a sequence of attribute or method names. (The order of the names is not important.) The *implicit* flag allows adapting objects that don't explicitly declare support for the protocol.

If you supply a true value for the *implicit* flag, the returned protocol will also adapt objects that have the specified methods or attributes. In other words, `protocolForType(file, ['read','close'], True)` returns a protocol that will consider any object with `read` and `close` methods to provide that protocol, as well as objects that explicitly support `protocolForType(file, ['read','close'])`.

A more detailed description of this function's operations and purpose may be found in section 1.1.8.

(Note: this function may generate up to `2**len(`*methods*`)` protocol objects, so beware of using large method lists.)

protocolForURI(*uri*)

New in version 0.9.1. Return a protocol object that represents the supplied URI or UUID string. It is guaranteed that you will receive the same protocol object if you call this routine more than once with equal strings. This behavior is preserved even across pickling and unpickling of the returned protocol object.

The purpose of this function is to permit modules to refer to protocols defined in another module, that may or may not be present at runtime. A more detailed description of this function's operations and purpose may be found in section 1.1.8.

`sequenceOf`(*protocol*)

New in version 0.9.1. Return a protocol object that represents a sequence of objects adapted to *protocol*. Thus, `protocols.sequenceOf(IFoo)` is a protocol that represents a `protocols.IBasicSequence` of objects supporting the `IFoo` protocol. It is guaranteed that you will receive the same protocol object if you call this routine more than once with the same protocol, even across pickling and unpickling of the returned protocol object.

When this function creates a new sequence protocol, it automatically declares an adapter function from `protocols.IBasicSequence` to the new protocol. The adapter function returns the equivalent of `[adapt(x,protocol) for x in sequence]`, unless one of the adaptations fails, in which case it returns `None`, causing the adaptation to fail.

The built-in `list` and `tuple` types are declared as implementations of `protocols.IBasicSequence`, so protocols returned by `sequenceOf()` can be used immediately to convert lists or tuples into lists of objects supporting *protocol*. If you need to adapt other kinds of sequences using your `sequenceProtocol()`, you will need to declare that those sequences implement `protocols.IBasicSequence` unless they subclass `tuple`, `list`, or some other type that implements `protocols.IBasicSequence`.

class `StickyAdapter`(*ob, proto*)

New in version 0.9.1. This base class is the same as the `Adapter` class, but with an extra feature. When a `StickyAdapter` instance is created, it declares itself as an adapter for its `subject`, so that subsequent `adapt()` calls will return the same adapter instance. (Technically, it declares an adapter function that returns itself.)

This approach is useful when an adapter wants to hold information on behalf of its subject, that must not be lost when the subject is adapted in more than one place.

Note that for a `StickyAdapter` subclass to be useful, the types it adapts *must* support `IOpenProvider`. See section 1.1.6, "Protocol Declarations for Individual Objects" for more information on this. Also, you should never declare that a `StickyAdapter` subclass adapts an individual object (as opposed to a type or protocol), since such a declaration would create a conflict when the adapter instance tries to register itself as an adapter for that same object and protocol.

`StickyAdapter` adds one attribute to those defined by `Adapter`:

`attachForProtocols`

A tuple of protocols to be declared by the constructor. Define this in your subclass' body to indicate what protocols it should attach itself for.

These classes and functions are also available from the top-level `protocols` package. In contrast to the items already covered, these classes and functions are generally needed only when extending the protocols framework, as opposed to merely using it.

class `Protocol`

`Protocol` is a base class that implements the `IOpenProtocol` interface, supplying internal adapter registries for adapting from other protocols or types/classes. Note that you do not necessarily need to use this class (or any other `IOpenProtocol` implementation) in your programs. Any object that implements the simpler `IProtocol` or `IAdaptingProtocol` interfaces may be used as protocols for the `adapt()` function. Compliance with the `IOpenProtocol` interface is only required to use the `protocols` declaration API. (That is, functions whose names begin with `declare` or `advise`.)

To create protocols dynamically, you can create individual `Protocol` instances, and then use them with the declaration API. You can also subclass `Protocol` to create your own protocol types. If you override `__init__`, however, be sure to call `Protocol.__init__()` in your subclass' `__init__` method.

class `Variation`(*baseProtocol* [, *context=None*])

New in version 0.9.1. A `Variation` is a `Protocol` that "inherits" adapter declarations from an existing protocol. When you create a `Variation`, it declares that it is implied by its *baseProtocol*, and so any adpater suitable for adapting to the base protocol is therefore suitable for the `Variation`. This allows you to then declare adapters to the variation protocol, without affecting those declared for the base protocol. In this way, you can have a protocol object that represents the use of the base protocol in a particular context. You can optionally specify that context via the *context* argument, which will then serve as the `context` attribute of the protocol. For more background on how this works and what it might be used for, see section 1.1.10.

class `AbstractBaseMeta`(*name, bases, dictionary*)

New in version 0.9.3. `AbstractBaseMeta`, a subclass of `Protocol` and `type`, is a metaclass used to create new "ABC-style" protocol objects, using class statements. You can use this metaclass directly, but it's generally simpler to just subclass `AbstractBase` instead. Normally, you will only use `AbstractBaseMeta` if you need to combine it with another metaclass.

class `InterfaceClass`(*name, bases, dictionary*)

`InterfaceClass` is a subclass of `AbstractBaseMeta` that implements the convenience adapation API (see section 1.1.3) for its instances. This metaclass is used to create new "pure-style" interfaces (i.e., protocol objects) using class statements. Normally, you will only use `InterfaceClass` directly if you need to combine it with another metaclass, as it is usually easier just to subclass `Interface`.

class `IBasicSequence`(*New in version .*)

0.9.1 This interface represents the ability to iterate over a container-like object, such as a list or tuple. An `IBasicSequence` object must have an `__iter__()` method. By default, only the built-in `list` and `tuple` types are declared as having instances providing this interface. If you want to be able to adapt to `sequenceOf()` protocols from other sequence types, you should declare that their instances support this protocol.

class `ProviderMixin`

If you have a class with a `__conform__` method for its instances, but you also want the instances to support `IOpenprovider` (so that `adviseObject` can be used on them), you may want to include this class as one of your class' bases. The default adapters for `IOpenprovider` can only adapt objects that do not already have a `__conform__` method of their own.

So, to support `IOpenprovider` with a custom `__conform__` method, subclass `ProviderMixin`, and have your `__conform__` method invoke the base `__conform__` method as a default, using `supermeta()`. (E.g. `return supermeta(MyClass,self).__conform__(protocol)`.) See below for more on the `supermeta()` function.

`metamethod`(*func*)

Wrap *func* in a manner analagous to `classmethod` or `staticmethod`, but as a metaclass-level method

that may be redefined by metaclass instances for their instances. For example, if a metaclass wants to define a `__conform__` method for its instances (i.e. classes), and those instances (classes) want to define a `__conform__` method for *their* instances, the metaclass should wrap its `__conform__` method with `metamethod`. Otherwise, the metaclass' `__conform__` method will be hidden by the class-level `__conform__` defined for the class' instances.

`supermeta`(*typ, ob*)

Emulates the Python built-in `super()` function, but with support for metamethods. If you ordinarily would use `super()`, but are calling a `metamethod`, you should use `supermeta()` instead. This is because Python 2.2 does not support using super with properties (which is effectively what metamethods are).

Note that if you are subclassing `ProviderMixin` or `Protocol`, you will need to use `supermeta()` to call almost any inherited methods, since most of the methods provided are wrapped with `metamethod()`.

`declareAdapterForType`(*protocol, adapter, typ* [, *depth=1*])

Declare that *adapter* adapts instances of class or type *typ* to *protocol*, by adapting *protocol* to `IOpenProtocol` and calling its `registerImplementation` method. If *typ* is adaptable to `IOpenImplementor`, its `declareClassImplements` method is called as well.

`declareAdapterForObject`(*protocol, adapter, ob* [, *depth=1*])

Declare that *adapter* adapts the object *ob* to *protocol*, by adapting *protocol* to `IOpenProtocol` and calling its `registerObject` method. Typically, *ob* must support `IOpenProvider`. See section 1.1.6 for details.

`declareAdapterForProtocol`(*protocol, adapter, proto* [, *depth=1*])

Declare that *adapter* adapts objects that provide protocol *proto* to *protocol*, by calling `adapt(`*proto*`,IOpenProtocol).addImpliedProtocol(`*protocol*`,`*adapter*`,`*depth*`)`.

Note: All of the interfaces listed here can also be imported directly from the top-level `protocols` package. However, you will probably only need them if you are extending the framework, as opposed to merely using it.

class `IOpenProtocol`

This interface documents the behavior required of protocol objects in order to be used with the `protocols` declaration API (the functions whose names begin with `declare` or `advise`.) The declaration API functions will attempt to `adapt()` supplied protocols to this interface.

The methods an `IOpenProtocol` implementation must supply are:

`addImpliedProtocol`(*proto, adapter, depth*)

Declare that this protocol can be adapted to protocol *proto* via the `IAdapterFactory` supplied in *adapter*, at the specified implication level *depth*. The protocol object should ensure that the implied protocol is able to adapt objects implementing its protocol (typically by recursively invoking `declareAdapterForType()` with increased depth and appropriately composed adapters), and notify any registered implication listeners via their `newProtocolImplied()` methods. If the protocol already implied *proto*, this method should have no effect and send no notifications unless the new *adapter* and *depth* represent a "shorter path" as described in section 1.1.7.

`registerImplementation`(*klass, adapter, depth*)

Declare that instances of type or class *klass* can be adapted to this protocol via the `IAdapterFactory` supplied in *adapter*, at the specified implication level *depth*. Unless *adapter* is `DOES_NOT_SUPPORT`, the protocol object must ensure that any protocols it implies are also able to perform the adaptation (typically by recursively invoking `declareAdapterForType()` with increased depth and appropriately composed adapters for its implied protocols). If the protocol already knew a way to adapt instances of *klass*, this method should be a no-op unless the new *adapter* and *depth* represent a "shorter path" as described in section 1.1.7.

`registerObject`(*ob, adapter, depth*)

Ensure that the specific object *ob* will be adapted to this protocol via the `IAdapterFactory` supplied in *adapter*, at the specified implication level *depth*. The protocol object must also ensure that the object can be adapted to any protocols it implies. This method may be implemented by adapting *ob* to `IOpenProvider`, calling the `declareProvides()` method, and then recursively invoking `declareAdapterForObject` with increased depth and appropriately composed adapters for the protocols' implied protocols.

`addImplicationListener`(*listener*)

Ensure that *listener* (an `IImplicationListener`) will be notified whenever an implied protocol is added to this protocol, or an implication path from this protocol is shortened. The protocol should at most retain a weak reference to *listener*. Note that if a protocol can guarantee that no notices will ever need to be sent, it is free to implement this method as a no-op. For example, Zope interfaces cannot imply any protocols besides their base interfaces, which are not allowed to change. Therefore, no change notifications would ever need to be sent, so the `IOpenProtocol` adapter for Zope interfaces implements this method as a no-op.

Note: `IOpenProtocol` is a subclass of `IAdaptingProtocol`, which means that implementations must therefore meet its requirements as well, such as having an `__adapt__()` method.

class `IOpenProvider`

This interface documents the behavior required of an object to be usable with `adviseObject()`. Note that some protocol objects, such as the `IOpenProtocol` adapter for Zope interfaces, can handle `adviseObject()` operations without adapting the target object to `IOpenProvider`. This should be considered an exception, rather than the rule. However, the `protocols` package declares default adapters so that virtually any Python object that doesn't already have a `__conform__()` method can be adapted to `IOpenProvider` automatically.

`declareProvides`(*protocol, adapter, depth*)

Declare that this object can provide *protocol* if adapted by the `IAdapterFactory` supplied in *adapter*,

at implication level *depth*. Return a true value if the new adapter was used, or a false value if the object already knew a "shorter path" for adapting to *protocol* (as described in section 1.1.7). Typically, an implementation of this method will also adapt *protocol* to `IOpenProtocol`, and then register with `addImplicationListener()` to receive notice of any protocols that might be implied by *protocol* in future.

class `IImplicationListener`

This interface documents the behavior required of an object supplied to the `IOpenProtocol.addImplicationListener()` method. Such objects must be weak-referenceable, usable as a dictionary key, and supply the following method:

`newProtocolImplied`(*srcProto, destProto, adapter, depth*)

Receive notice that an adaptation was declared from *srcProto* to *destProto*, using the `IAdapterFactory` *adapter*, at implication level *depth*.

When used as part of an `IOpenProvider` implementation, this method is typically used to recursively invoke `declareAdapterForObject()` with increased depth and appropriately composed adapters from protocols already supported by the object.

class `IOpenImplementor`

If an class or type supplied to `declareAdapterForType` supports this interface, it will be notified of the declaration and any future declarations that affect the class, due to current or future protocol implication relationships. Supporting this interface is not necessary; it is provided as a hook for advanced users. Note that to declare a class or type as an `IOpenImplementor`, you must call `adviseObject(theClass, provides=[IOpenImplementor])` after the class definition or place `advise(classProvides=[IOpenImplementor])` in the body of the class, since this interface must be provided by the class itself, not by its instances. (Of course, if you implement this interface via a metaclass, you can declare that the metaclass' instances provide the interface.)

Notification to classes supporting `IOpenImplementor` occurs via the following method:

`declareClassImplements`(*protocol, adapter, depth*)

Receive notice that instances of the class support *protocol* via the the `IAdapterFactory` supplied in *adapter*, at implication level *depth*.

class `IAdapterFactory`

An interface documenting the requirements for an object to be used as an adapter factory: i.e., that it be a callable accepting an object to be adapted.) This interface is not used by the `protocols` package except as documentation.

class `IProtocol`

An interface documenting the basic requirements for an object to be used as a protocol for `adapt()`: i.e., that it be usable as a dictionary key. This interface is not used by the `protocols` package except as documentation.

class `IAdaptingProtocol`

An interface documenting the requirements for a protocol object to be able to adapt objects when used with `adapt()`: i.e., that it have a `__adapt__` method that accepts the object to be adapted and returns either an object providing the protocol or `None`. This interface is not used by the `protocols` package except as documentation. It is a subclass of `IProtocol`, so any implementation of this interface must support the requirements defined by `IProtocol` as well.

`protocols.adapters` — "Adapter arithmetic" support

The `protocols.adapters` module provides support for doing "adapter arithmetic" such as determining which of two adapter paths is shorter, composing a new adapter from two existing adapters, and updating an adapter registry with a new adapter path. See section 1.1.7 for a more general discussion of adapter arithmetic.

`minimumAdapter`(*a1, a2* [, *d1=0, d2=0*])

Find the "shortest" adapter path, *a1* at depth *d1*, or *a2* at depth *d2*. Assuming *a1* and *a2* are adapter factories that accept similar input and return similar output, this function returns the one which is the "shortest path" between its input and its output. That is, the one with the smallest implication depth (*d1* or *d2*), or, if the depths are equal, then the adapter factory that is composed of the fewest chained factories (as composed by `composeAdapters()`) is returned. If neither factory is composed of multiple factories, or they are composed of the same number of intermediate adapter factories, then the following preference order is used:

1. If one of the adapters is `NO_ADAPTER_NEEDED`, it is returned
2. If one of the adapters is `DOES_NOT_SUPPORT`, the *other* adapter is returned.
3. If both adapters are the exact same object (i.e. `a1 is a2`), either one is returned

If none of the above conditions apply, then the adapter precedence is considered ambiguous, and a `TypeError` is raised.

This function is used by `updateWithSimplestAdapter` to determine whether a new adapter declaration should result in a registry update. Note that the determination of adapter composition length uses the `__adapterCount__` attribute, if present. (It is assumed to be `1` if not present. See `composeAdapters()` for more details.)

`composeAdapters`(*baseAdapter, baseProtocol, extendingAdapter*)

Return a new `IAdapterFactory` composed of the input adapter factories *baseAdapter* and *extendingAdapter*. If either input adapter is `DOES_NOT_SUPPORT`, `DOES_NOT_SUPPORT` is returned. If either input adapter is `NO_ADAPTER_NEEDED`, the other input adapter is returned. Otherwise, a new adapter factory is created that will return *extendingAdapter*`(`*baseAdapter*`(object))` when called with `object`. (Note: the actual implementation verifies that *baseAdapter* didn't return `None` before it calls *extendingAdapter*).

If this function creates a new adapter factory, the factory will have an `__adapterCount__` attribute set to the sum of the `__adapterCount__` attributes of the input adapter factories. If an input factory does not have an `__adapterCount__` attribute, it is assumed to equal `1`. This is done so that the `minimumAdapter()` can compare the length of composed adapter chains.

`updateWithSimplestAdapter`(*mapping, key, adapter, depth*)

Treat *mapping* as an adapter registry, replacing the entry designated by *key* with an `(`*adapter*`,`*depth*`)` tuple, if and only if the new entry would be a "shorter path" than the existing entry, if any. (I.e., if `minimumAdapter(old,` *adapter*`, oldDepth,` *depth*`)` returns *adapter*, and *adapter* is not the existing registered adapter. The function returns a true value if it updates the contents of *mapping*.

This function is used to manage type-to-protocol, protocol-to-protocol, and object-to-protocol adapter registries, keyed by type or protocol. The *mapping* argument must be a mapping providing `__setitem__()` and `get()` methods. Values stored in the mapping will be `(`*adapter*`,`*depth*`)` tuples.

protocols.zope_support — Support for Zope Interfaces

Importing this module enables experimental support for using Zope X3 `Interface` objects with the `protocols` package, by registering an adapter from Zope X3's `InterfaceClass` to `IOpenProtocol`. The adapter supports the following subset of the declaration API:

- The only adapters supported via Zope APIs are `NO_ADAPTER_NEEDED` and `DOES_NOT_SUPPORT`. By using PyProtocols APIs, you may declare and use other adapters for Zope interfaces, but Zope itself will not use them, since the Zope interface API does not directly support adaptation.
- Zope's interface APIs do not conform to `protocols` package "shortest path wins" semantics. Instead, new declarations override older ones.
- Interface-to-interface adaptation may not work if a class only declares what it implements using Zope's interface API. That is, if a class declares that it implements `ISomeZopeInterface`, and you define an adaptation from `ISomeZopeInterface` to `ISomeOtherInterface`, PyProtocols may not recognize that the class can be adapted to `ISomeOtherInterface`.
- Changing the `__bases__` of a class that has Zope interfaces declared for it (either as "class provides" or "instances provide"), may have unexpected results, because Zope uses inheritance of a single descriptor to control declarations. In general, it will only work if the class whose `bases` are changed, has no declarations of its own.
- You cannot declare an implication relationship from a Zope `Interface`, because Zope only supports implication via inheritance, which is fixed at interface definition time. Therefore, you cannot create a "subset" of a Zope `Interface`, and subscribing an `IImplicationListener` to an adapted Zope `Interface` silently does nothing.
- You can, however, declare that a `protocols.Interface` extends a Zope `Interface`. Declaring that a class' instances or that an object provides the extended interface, will automatically declare that the class' instances or the object provides the Zope `Interface` as well. For example:

```
import protocols
from zope.somepackage.interfaces import IBase

class IExtended(protocols.Interface):
    advise(
        protocolExtends = [IBase]
    )

class AnImplementation:
    advise(
        instancesProvide = [IExtended]
    )
```

  The above code should result in Zope recognizing that instances of `AnImplementation` provide the Zope `IBase` interface.

- You cannot extend both a Zope interface and a Twisted interface in the same `protocols.Interface`. Although this may not give you any errors, Twisted and Zope both expect to use an `__implements__` attribute to store information about what interface a class or object provides. But each has a different interpretation of the contents, and does not expect to find "foreign" interfaces contained within. So, until this issue between Zope and Twisted is resolved, it is not very useful to create interfaces that extend both Zope and Twisted interfaces.
- Zope does not currently appear to support classes inheriting direct declarations (e.g. `classProvides`). This appears to be a by-design limitation.

The current implementation of support for Zope X3 interfaces is currently based on Zope X3 beta 1; it may not work with older releases. Zope X3 requires Python 2.3.4 or better, so even though PyProtocols works with 2.2.2 and up in general, you will need 2.3.4 to use PyProtocols with Zope X3.

protocols.twisted_support — Support for Twisted Interfaces

Importing this module enables experimental support for using Twisted 1.1.0 `Interface` objects with the `protocols` package, by registering an adapter from Twisted's `MetaInterface` to `IOpenProtocol`. The adapter supports the following subset of the declaration API:

- Only protocol-to-protocol adapters defined via the `protocols` declaration API will be available to implication listeners. If protocol-to-protocol adapters are registered via Twisted's `registerAdapter()`, implication listeners are *not* notified.
- You cannot usefully create a "subset" of a Twisted interface, or an adaptation from a Twisted interface to another interface type, as Twisted insists that interfaces must subclass its interface base class. Also, Twisted does not support transitive adaptation, nor can it notify the destination interface(s) of any new incoming adapter paths.
- If you register an adapter factory that can return `None` with a Twisted interface, note that Twisted does not check for a `None` return value from `getAdapter()`. This means that code in Twisted might receive `None` when it expected either an implementation or an error.
- Only Twisted's global adapter registry is supported for declarations and `adapt()`.
- Twisted doesn't support classes providing interfaces (as opposed to their instances providing them). You may therefore obtain unexpected results if you declare that a class provides a Twisted interface or an interface that extends a Twisted interface.
- Changing the `__bases__` of a class that has Twisted interfaces declared for it may have unexpected results, because Twisted uses inheritance of a single descriptor to control declarations. In general, it will only work if the class whose `bases` are changed, has no declarations of its own.
- Any adapter factory may be used for protocol-to-protocol adapter declarations. But, for any other kind of declaration, `NO_ADAPTER_NEEDED` and `DOES_NOT_SUPPORT` are the only adapter factories that can be used with Twisted.
- Twisted interfaces do not conform to `protocols` package "shortest path wins" semantics. For protocol-to-protocol adapter declarations, only one adapter declaration between a given pair of interfaces is allowed. Any subsequent declarations with the same source and destination will result in a `ValueError`. For all other kinds of adapter declarations, new declarations override older ones.
- You cannot extend both a Zope interface and a Twisted interface in the same `protocols.Interface`. Although this may not give you any errors, Twisted and Zope both expect to use an `__implements__` attribute to store information about what interface a class or object provides. But each has a different interpretation of the contents, and does not expect to find "foreign" interfaces contained within. So, until this issue between Zope and Twisted is resolved, it is not very useful to create interfaces that extend both Zope and Twisted interfaces.

protocols.advice — Metaclasses and other "Magic"

This module provides a variety of utility functions and classes used by the `protocols` package. None of them are really specific to the `protocols` package, and so may be useful to other libraries or applications.

`addClassAdvisor`(*callback* [, *depth=2*])

Set up *callback* to be called with the containing class, once it is created. This function is designed to be called by an "advising" function (such as `protocols.advise()`) executed in the body of a class suite. The "advising" function supplies a callback that it wishes to have executed when the containing class is created. The callback will be given one argument: the newly created containing class. The return value of the callback will be used in *place* of the class, so the callback should return the input if it does not wish to replace the class.

The optional *depth* argument determines the number of frames between this function and the targeted class suite. *depth* defaults to 2, since this skips this function's frame and one calling function frame. If you use this function from a function called directly in the class suite, the default will be correct, otherwise you will need to determine the correct depth yourself.

This function works by installing a special class factory function in place of the `__metaclass__` of the containing class. Therefore, only callbacks *after* the last `__metaclass__` assignment in the containing class will be executed. Be sure that classes using "advising" functions declare any `__metaclass__` *first*, to ensure all callbacks are run.

`isClassAdvisor`(*ob*)

Returns truth if *ob* is a class advisor function. This is used to determine if a `__metaclass__` value is a "magic" metaclass installed by `addClassAdvisor()`. If so, then *ob* will have a `previousMetaclass` attribute pointing to the previous metaclass, if any, and a `callback` attribute containing the callback that was given to `addClassAdvisor()`.

`getFrameInfo`(*frame*)

Return a `(`*kind*`,`*module*`,`*locals*`,`*globals*`)` tuple for the supplied frame object. The returned *kind* is a string: either "exec", "module", "class", "function call", or "unknown". *module* is the module object the frame is/was executed in, or `None` if the frame's globals could not be correlated with a module in `sys.modules`. *locals* and *globals* are the frame's local and global dictionaries, respectively. Note that they can be the same dictionary, and that modifications to locals may not have any effect on the execution of the frame.

This function is used by functions like `addClassAdvisor()` and `advise()` to verify where they're being called from, and to work their respective magics.

`getMRO`(*ob* [, *extendedClassic=False*])

Return an iterable over the "method resolution order" of *ob*. If *ob* is a "new-style" class or type, this returns its `__mro__` attribute. If *ob* is a "classic" class, this returns `classicMRO(`*ob*`,`*extendedClassic*`)`. If *ob* is not a class or type of any kind, a one-element sequence containing just *ob* is returned.

`classicMRO`(*ob* [, *extendedClassic=False*])

Return an iterator over the "method resolution order" of classic class *ob*, following the "classic" method resolution algorithm of recursively traversing `__bases__` from left to right. (Note that this may return the same class more than once, for some inheritance graphs.) If varextendedClassic is a true value, `InstanceType` and `object` are added at the end of the iteration. This is used by `Protocol` objects to allow generic adapters for `InstanceType` and `object` to be used with "classic" class instances.

`determineMetaclass`(*bases* [, *explicit_mc=None*])

Determine the metaclass that would be used by Python, given a non-empty sequence of base classes, and an optional explicitly supplied `__metaclass__`. Returns `ClassType` if all bases are "classic" and there is no *explicit_mc*. Raises `TypeError` if the bases' metaclasses are incompatible, just like Python would.

`minimalBases`(*classes*)

Return the shortest ordered subset of the input sequence *classes* that still contains the "most specific" classes. That is, the result sequence contains only classes that are not subclasses of each other. This function is used by `determineMetaclass()` to narrow down its list of candidate metaclasses, but is also useful for dynamically generating metaclasses.

mkRef(*ob* [, *callable*])

If *ob* is weak-referenceable, returns `weakref.ref(`*ob*`,`*callable*`)`. Otherwise, returns a `StrongRef(`*ob*`)`, emulating the interface of `weakref.ref()`. This is used by code that wants to use weak references, but may be given objects that are not weak-referenceable. Note that *callable*, if supplied, will *not* be called if *ob* is not weak-referenceable.

class StrongRef(*ob*)

An object that emulates the interface of `weakref.ref()`. When called, an instance of `StrongRef` will return the *ob* it was created for. Also, it will hash the same as *ob* and compare equal to it. Thus, it can be used as a dictionary key, as long as the underlying object can. Of course, since it is not really a weak reference, it does not contribute to the garbage collection of the underlying object, and may in fact hinder it, since it holds a live reference to the object.

1.1.10 Big Example 2 — Extending the Framework for Context

Now it's time for our second "big" example. This time, we're going to add an extension to the `protocols` framework to support "contextual adaptation". The tools we've covered so far are probably adequate to support 80-90% of situations requiring adaptation. But, they are essentially global in nature: only one adapter path is allowed between any two points. What if we need to define a different adaptation in a specific context?

For example, let's take the documentation framework we began designing in section 1.1.1. Suppose we'd like, for the duration of a single documentation run, to replace the factory that adapts from `FunctionType` to `IDocumentable`? For example, we might like to do this so that functions used by our "finite state machine" objects as "transitions" are documented differently than regular functions.

Using only the tools described so far, we can't do this if `IDocumentable` is a single object. The framework that registered the `FunctionAsDocumentable` adapter effectively ensured that we cannot replace that adapter with another, since it is already the shortest adapter path. What can we do?

In section 1.1.7, we discussed how we could create "subset" protocols and "inherit" adapter declarations from existing protocols. In this way, we could create a new subset protocol of `IDocumentable`, and then register our context-specific adapters with that subset. These subset protocols are just as fast as the original protocols in looking up adapters, so there's no performance penalty.

But who creates the subset protocol? The client or the framework? And how do we get the framework to use our subset instead of its built-in `IDocumentable` protocol?

To answer these questions, we will create an extension to the `protocols` framework that makes it easy for frameworks to manage "contextual" or "local" protocols. Then, framework creators will have a straightforward way to support context-specific adapter overrides.

As before, we'll start by envisioning our ideal situation. Let's assume that our documentation tools are object-based. That is, we instantiate a "documentation set" or "documentation run" object in order to generate documentation. How do we want to register adapters? Well, we could have the framework add a bunch of methods to do this, but it seems more straightforward to simply supply the interfaces as attributes of the "documentation set" or "documentation run" object, e.g.:

```
from theDocTool import DocSet
from myAdapters import specialFunctionAdapter
from types import FunctionType
import protocols

myDocs = DocSet()

protocols.registerAdapter(
    specialFunctionAdapter,
    provides = [myDocs.IDocumentable],
    forTypes = [FunctionType]
)

myDocs.run()
```

So, instead of importing the interface, we access it as an attribute of some relevant "context" object, and declare adapters for it. Anything we don't declare a "local" adapter for, will use the adapters declared for the underlying "global" protocol.

Naturally, the framework author could implement this by writing code in the `DocSet` class' `__init__` method, to create the new "local" protocol and register it as a subset of the "global" `IDocumentable` interface. But that would be time-consuming and error prone, and therefore discourage the use of such "local" protocols.

Again, let's consider what our ideal situation would be. The author of the `DocSet` class should be able to do something like:

```
class DocSet:

    from doctool.interfaces import IDocumentable, ISignature

    IDocumentable = subsetPerInstance(IDocumentable)
    ISignature = subsetPerInstance(ISignature)

    # ... etc.
```

Our hypothetical `subsetPerInstance` class would be a descriptor that did all the work needed to provide a "localized" version of each interface for each instance of `DocSet`. Code in the `DocSet` class would always refer to `self.IDocumentable` or `self.ISignature`, rather than using the "global" versions of the interfaces. Thus, we can now register adapters that are unique to a specific `DocSet`, but still use any globally declared adapters as defaults.

Okay, so that's our hypothetical ideal. How do we implement it? I personally like to try writing the ideal thing, to find out what other pieces are needed. So let's start with writing the `subsetPerInstance` descriptor, since that's really the only piece we know we need so far.

```
from protocols import Protocol, declareAdapterForProtocol, NO_ADAPTER_NEEDED

class subsetPerInstance(object):

    def __init__(self,protocol,name=None):

        self.protocol = protocol
        self.name = name or getattr(protocol,'__name__',None)

        if not self.name:
            raise TypeError("Descriptor needs a name for", protocol)


    def __get__(self,ob,typ=None):

        if ob is None:
            return self

        name = self.name
        if getattr(type(ob),name) is not self or name in ob.__dict__:
            raise TypeError(
                "Descriptor is under more than one name or the wrong name",
                self, name, type(ob)
            )

        local = Protocol()
        declareAdapterForProtocol(local,NO_ADAPTER_NEEDED,self.protocol)

        # save it in the instance's dictionary so we won't be called again
        ob.__dict__[name] = local
        return local


    def __repr__(self):
        return "subsetPerInstance(%r)" % self.protocol
```

Whew. Most of the complexity above comes from the need for the descriptor to know its "name" in the containing class. As written, it will guess its name to be the name of the wrapped interface, if available. It can also detect some

potential aliasing/renaming issues that could occur. The actual work of the descriptor occurs in just two lines, buried deep in the middle of the `__get__` method.

As written, it's a handy enough tool. We could leave things where they are right now and still get the job done. But that would hardly be an example of extending the framework, since we didn't even subclass anything!

So let's add another feature. As it sits, our descriptor should work with both old and new-style classes, automatically generating one subset protocol for each instance of its containing class. But, the subset protocol doesn't *know* it's a subset protocol, or of what context. If we were to print `DocSet().IDocumentable`, we'd just get something like `<protocols.interfaces.Protocol instance at 0x00ABA220>`.

Here's what we'd like it to do instead. We'd like it to say something like `LocalProtocol(<class 'IDocumentable'>, <DocSet instance at 0x00AD9FB0>)`. That is, we want the local protocol to:

- "know" it's a local protocol
- know what protocol it's a local subset of
- know what "context" object it's a local protocol for

What does this do for us? Aside from debugging, it gives us a chance to find related interfaces, or access methods or data available from the context.

So, let's create a `LocalProtocol` class:

```
class LocalProtocol(Protocol):

    def __init__(self, baseProtocol, context):

        self.baseProtocol = baseProtocol
        self.context = context

        # Note: Protocol is a ''classic'' class, so we don't use super()
        Protocol.__init__(self)

        declareAdapterForProtocol(self,NO_ADAPTER_NEEDED,baseProtocol)

    def __repr__(self):
        return "LocalProtocol(%r,%r)" % (self.baseProtocol, self.context)
```

And now, we can replace these two lines in our earlier `__get__` method:

```
            local = Protocol()
            declareAdapterForProtocol(local,NO_ADAPTER_NEEDED,self.protocol)
```

with this one:

```
            local = LocalProtocol(self.protocol, ob)
```

Thus, the new local protocol will know its context is the instance it was retrieved from.

Of course, to make this new extension really robust, we would need to add some more documentation. For example, it might be good to add an `ILocalProtocol` interface that documents what local protocols do. Context-sensitive adapters would then be able to verify whether they are working with a local protocol or a global one. Framework developers would also want to document what local interfaces are provided by their frameworks' objects, and authors of context-sensitive adapters need to document what interface they expect their local protocols' `context` attribute to supply! Also, see below for a web site with some interesting papers on patterns for using localized adaptation of this kind.

Note: In practice, the idea of having local protocols turned out to be useful enough that as of version 0.9.1, our `LocalProtocol` example class was added to the protocols package as `protocols.Variation`. So, if you want to make use of the idea, you don't need to type in the source or write your own any more.

See Also:

Object Teams
(http://www.objectteams.org/)

If you find the idea of context-specific interfaces and adapters interesting, you'll find "Object Teams" intriguing as well. In effect, the ideas we've presented here map onto a subset of the "Object Teams" concept. Our local interfaces correspond to their "abstract roles", our local adapters' instances map to their "role instances", and our contexts are their "team instances". Adapting an object corresponds to their "lifting", and so on. The main concept that's not directly supported by our implementation here is "callin binding". (Callin binding is a way of (possibly temporarily) injecting hooks into an adapted object so that the adapter can be informed when the adapted object's methods are called directly by other code.)

1.1.11 Additional Examples and Usage Notes

If you have any ideas or examples you'd like to share for inclusion in this section, please contact the author. In the meantime, here are a few additional examples of things you can do with `adapt()` and the `protocols` package.

Double Dispatch and the "Visitor" Pattern

Double dispatch and the "Visitor" pattern are mechanisms for selecting a method to be executed, based on the type of two objects at the same time. To implement either pattern, both object types must have code specifically to support the pattern. Object adaptation makes this easier by requiring at most one of the objects to directly support the pattern; the other side can provide support via adaptation. This is useful both for writing new code clearly and for adapting existing code to use the pattern.

First, let's look at double dispatching. Suppose we are creating a business application GUI that supports drag-and-drop. We have various kinds of objects that can be dragged and dropped onto other objects: users, files, folders, a trash can, and a printer. When we drop a user on a file, we want to grant the user access to the file, and when we drop a file on the user, we want to email them the file. If we drop a file on a folder, it should be filed in the folder, but if we drop the folder on the file, that's an error. The classic "double dispatch" approach would look something like:

```
class Printer:
    def drop(self,thing):
        thing.droppedOnPrinter(self)

class Trashcan:
    def drop(self,thing):
        thing.droppedInTrash(self)

class User:
    def drop(self,thing):
        thing.droppedOnUser(self)

    def droppedOnPrinter(self,printer):
        printer.printUser(self)

    def droppedInTrash(self,trash):
        self.delete()

class File:
    def drop(self,thing):
        thing.droppedOnFile(self)

    def droppedOnPrinter(self,printer):
        printer.printFile(self)

    def droppedOnUser(self,user):
        user.sendMail(self)

    def droppedInTrash(self,trash):
        self.delete()
```

We've left out any of the methods that actually *do* anything, of course, and all of the methods for things that the objects don't do. For example, the `Trashcan` should have methods for `droppedInTrash()`, `droppedOnPrinter()`, etc., that display an error or beep or whatever. (Of course, in Python you can just trap the `AttributeError` from the missing method to do this; but we didn't show that here either.)

Every time another kind of object is added to this system, new `droppedOnX` methods spring up everywhere like

weeds. Now let's look at the adaptation approach:

```
class Printer:
    def drop(self,thing):
        IPrintable(thing).printOn(self)

class Trashcan:
    def drop(self,thing):
        IDeletable(thing).delete(self)

class User:
    protocols.advise(instancesProvide=[IDeletable,IPrintable])
    def drop(self,thing):
        IMailable(thing).mailTo(self)

class File:
    protocols.advise(instancesProvide=[IDeletable,IMailable,IPrintable])
    def drop(self,thing):
        IInsertable(thing).insertInto(self)

class Undroppable(protocols.Adapter):
    protocols.advise(
        instancesProvide=[IPrintable,IDeletable,IMailable,IInsertable],
        asAdapterForTypes=[object]
    )

    def printOn(self,printer):
        print "Can't print", self.subject

    def mailTo(self,user):
        print "Can't mail", self.subject

    # ... etc.
```

Notice how our default `Undroppable` adapter class implements the `IPrintable`, `IDeletable`, `IMailable`, and `IInsertable` protocols on behalf of arbitrary objects, by giving user feedback that the operation isn't possible. (This technique of using a default adapter factory that provides an empty or error-raising implementation of an interface, is an example of the **null object pattern**.)

Notice that the adaptation approach is much more scalable, because new methods are not required for every new droppable item. Third parties can declare adaptations between two other developers' objects, making drag and drop between them possible.

Now let's look at the "Visitor" pattern. The "Visitor" pattern is a specialized form of double dispatch, used to apply an algorithm to a structured collection of objects. For example, the Python **docutils** tookit implements the visitor pattern to create various kinds of output from a document node tree (much like an XML DOM). Each node has a `walk()` method that accepts a "visitor" argument. The visitor must provide a set of `visit_X` methods, where X is the name of a type of node. The idea of the approach is that one can write new visitor types that perform different functions. One visitor writes out HTML, another writes out LaTeX or maybe plain ASCII text. The nodes don't care what the visitor does, they just tell it what kind of object is being visited.

Like double dispatch, this pattern is definitely an improvement over writing large if-then-else blocks to introspect types. But it does have a few drawbacks. First, all the types must have unique names. Second, the visitor must have methods for all possible node types (or the caller must handle the absence of the methods). Third, there is no way for the methods to mimic the inheritance or interface structure of the source types. So, if there are node types like `Shape` and `Square`, you must write `visit_Shape` and `visit_Square` methods, even if you would like to treat all subtypes of `Shape` the same.

The object adaptation approach to this, is to define visitor(s) as adapters from the objects being traversed, to an interface that supplies the desired behavior. For example, one might define `IHTMLWriter` and `ILaTeXWriter` interfaces, with `writeHTML()` and `writeLaTeX()` methods. Then, by defining adapters from the appropriate node base types to these interfaces, the desired behavior is achieved. Just use `IHTMLWriter(document).writeHTML()`, and off you go.

This approach is far less fragile, since new node types do not require new methods in the visitors, and if the new node type specializes an existing type, the default adaptation might be reasonable. Also, the approach is non-invasive, so it can be applied to existing frameworks that don't support the visitor pattern (such as `xml.dom.minidom`). Further, the adapters can exercise fine-grained control over any traversal that takes place, since it is the adapter rather than the adaptee that controls the visiting order.

Last, but not least, notice that by adapting from interfaces rather than types, one can apply this pattern to multiple implementations of the interface. For example, Python has many XML DOM implementations; to the extent that two implementations provide the same interface, the adapters you write could be used with any of them, even if each pacakge has different names for their node types.

Are there any downsides to using adaptation over double-dispatch or the Visitor pattern? The total size of your program may be larger, because you'll be writing lots of adapter classes. But, your program will also be more modular, and you'll be able to group the classes in ways that make more sense for the reader. Using adaptation also may be faster or slower than not using it, depending on various implementation factors.

It's rare that the difference is significant, however. In most uses of these patterns, runtime is dominated by the useful work being done, not by the dispatching. The exception is when a structure to be visited contains many thousands of elements that need virtually no work done to them. (For example, if an XML visitor wrote text nodes out unchanged, and the input was mostly text nodes.) Under such conditions, the time taken by the dispatch mechanism (whether name-based or adapter-based) would be more visible.

The author has found, however, that in that situation, one can gain more speed by registering null adapter factories or `DOES_NOT_SUPPORT` for the element types in question. This shortens the adapter lookup time enough to make the adaptation overhead competitive with name-based approaches. But this only needs to be done when "trivial" elements dominate the structures to be processed, *and* performance is critical.

See Also:

Variations on the Visitor Pattern
(http://citeseer.nj.nec.com/nordberg96variations.html)

> A critique of the Visitor pattern that raises some of the same issues we raise here, and with similar solutions. Reading its C++ examples, however, will increase your appreciation for the simplicity and modularity that `adapt()` offers!

The Null Object Pattern
(http://citeseer.nj.nec.com/woolf96null.html)

> The original write-up of this handy approach to simplifying framework code.

Replacing introspection with Adaptation, Revisited

> "Potentially-idempotent adapter functions are a honking great idea – let's do more of those", to paraphrase the timbot.
>
> — Alex Martelli, on comp.lang.python

All programs that use GOTO's can be rewritten without GOTOs, using higher-level constructs for control flow like function calls, `while` loops, and so on. In the same way, type checks – and even interface checks – are not essential in the presence of higher-level control constructs such as adaptation. Just as getting rid of GOTO "spaghetti code" helped make programs easier to read and understand, so too can replacing introspection with adaptation.

In section 1.1.3, we listed three common uses for using type or interface checks (e.g. using `isinstance()`):

- To manually adapt a supplied component to a needed interface
- To select one of several possible behaviors, based on the kind of component supplied
- To select another component, or take some action, using information about the interfaces supported by the supplied component

By now, you've seen enough uses of the `protocols` module that it should be apparent that all three of the above use cases can – in principle – be handled by adaptation. However, in the course of moving PEAK from using introspection to adaptation, I ran into some use cases that at first seemed very difficult to "adapt". However, once I understood how to handle them, I realized that there was a straightforward approach to refactoring any introspection use cases I encountered. Although this approach seems to have more than one step, in reality they are all variations on the same theme: expose the hidden interface, then adapt to it. Here's how you do it:

1. First, is this just a case of adapting different types to a common interface? If yes, then just declare the adapters and use `adapt()` normally. If the interface isn't explicit or documented, make it so.
2. Is this a case of choosing a behavior, based on the type? If yes, then *define the missing interface* that you want to adapt to. In other words, code that switches on type to select a behavior, really wants the behavior to be in the *other* object. So, there is in effect an "undocumented implicit interface" that the code is adapting the other object to. Make the interface explicit and documented, move the code into adapters (or into the other classes!), and use `adapt()`.
3. Is this a case of choosing a behavior or a component based on using interfaces as metadata? If so, this is really a special case of #2. An example of this use case is where Zope X3 provides UI components based on what interfaces an object supports. In this case, the "undocumented implicit interface" is the ability to select an appropriate UI component! Or perhaps it's an ability to provide a set of "tags" or "keys" that can be used to look up UI components or other things. You'll have to decide what the real "essence" is. But either way, you make the needed behavior explicit (as an interface), and then use `adapt()`.

Notice that in each case, the code is demonstrably improved. First, there is more documentation of the *intended* behavior (as opposed to merely the actual behavior, which might be broken). Second, there is greater extensibility, because it isn't necessary to change the code to add more type cases. Third, the code is more readable, because the code's purpose is highlighted, not all the possible variations of its implementation. In the words of Tim Peters, "Explicit is better than implicit. Simple is better than complex. Sparse is better than dense. Readability counts."

Now that we've covered how to replace all forms of introspection with adaptation, I'll readily admit that I still write code that does introspection when I'm in "first draft" mode! Brevity is the soul of prototyping, and I don't mind banging out a few quick `if isinstance():` checks in order to figure out what it is I want the code to do. But then, I refactor, because I want my code to be... adaptable! Chances are good, that you will too.

MODULE INDEX

P

INDEX

R

S

U

V