Reg Documentation

Release 0.6

Martijn Faassen

June 17, 2014

Contents

1 Using Reg **3**
- 1.1 Introduction . . . 3
- 1.2 Example . . . 3

2 Generic functions **5**
- 2.1 A Hypothetical CMS . . . 5
- 2.2 `size` methods . . . 5
- 2.3 Adding `size` from outside . . . 6
- 2.4 Generic size . . . 7
- 2.5 New `File` content . . . 8
- 2.6 New `HtmlDocument` content . . . 8
- 2.7 Doing this with Reg . . . 9
- 2.8 Using classes . . . 10
- 2.9 Multiple dispatch . . . 12

3 Service Discovery **15**

4 Lower level API **17**
- 4.1 Registering non-functions . . . 17
- 4.2 Getting all . . . 17
- 4.3 Using the Registry directly . . . 18

5 Composition **19**
- 5.1 ClassRegistry . . . 19
- 5.2 Caching . . . 20
- 5.3 Composing class lookups . . . 20

6 API **23**

7 Developing Reg **29**
- 7.1 Install Reg for development . . . 29
- 7.2 Running the tests . . . 29
- 7.3 Running the documentation tests . . . 30
- 7.4 Building the HTML documentation . . . 30
- 7.5 Various checking tools . . . 30

8 History of Reg **31**
- 8.1 Reg History . . . 31
- 8.2 Brief history of Zope Component Architecture . . . 32

9 Indices and tables 33

Python Module Index 35

Reg is a Python library that provides generic function support to Python. It help you build powerful registration and configuration APIs for your application, library or framework.

CHAPTER 1

Using Reg

1.1 Introduction

Reg lets you write generic functions. To support this, Reg provides an implementation of multiple dispatch in Python. Reg lets you define methods outside their classes as plain Python functions. Reg in its basic use is like the single dispatch implementation described in Python PEP 443, but Reg provides a lot more flexibility.

Reg supports loose coupling. You can define a function in your core application or framework but provide definitions of this function outside of it.

Reg gives developers fine control over how to find implemenations of these functions. You can have multiple independent dispatch registries, and you can also compose them together. For special use cases you can also register and look up other objects instead of functions.

What is Reg for? Reg offers infrastructure that lets you build more powerful frameworks – frameworks that can be extended and overridden in a general way. Reg may seem like overkill to you. You may very well be right; it depends on what you're building.

1.2 Example

Here is an example of Reg. First we define a generic function:

```
import reg
@reg.generic
def title(obj):
    return "we don't know the title"
```

We now create a few example classes. We want to be able to get the title for both.

```
class TitledReport(object):
    def __init__(self, title):
        self.title = title

class LabeledReport(object):
    def __init__(self, label):
        self.label = label
```

In one case there's an attribute called `title` but in the other case we have an attribute `label` we want to use as the title. We will implement this behavior in a few plain python functions:

```
def titled_report_title(obj):
    return obj.title

def labeled_report_title(obj):
    return obj.label
```

We now create a Reg `reg.Registry`, register our implementations in it using `reg.IRegistry.register()`, and then tell Reg to use it automatically using `reg.implicit.Implicit.initialize()`:

```
registry = reg.Registry()
registry.register(title, [TitledReport], titled_report_title)
registry.register(title, [LabeledReport], labeled_report_title)
from reg import implicit
implicit.initialize(registry)
```

Once we've done this, our generic `title` function works on both titled and labeled objects:

```
>>> titled = TitledReport('titled')
>>> labeled = LabeledReport('labeled')
>>> title(titled)
'titled'
>>> title(labeled)
'labeled'
```

Our example is over, so we reset the implicit registry set up before:

```
implicit.clear()
```

Why not just use plain functions or methods instead of generic functions? Often plain functions or methods will be the right solution. But not always – in this document we will motivate a case where generic functions are useful.

CHAPTER 2

Generic functions

2.1 A Hypothetical CMS

Let's look at how Reg works within the context of a hypothetical content management system (CMS).

This hypothetical CMS has two kinds of content item (we'll add more later):

- a `Document` which contains some text.
- a `Folder` which contains a bunch of content items, for instance `Document` instances.

This is the implementation of our CMS:

```
class Document(object):
    def __init__(self, text):
        self.text = text

class Folder(object):
    def __init__(self, items):
        self.items = items
```

2.2 `size` methods

Now we want to add a feature to our CMS: we want the ability to calculate the size (in bytes) of any content item. The size of the document is defined as the length of its text, and the size of the folder is defined as the sum of the size of everything in it.

`len(text)` is not in bytes!

Yeah, we're lying here. `len(text)` is not in bytes if text is in unicode. Just pretend that text is in ASCII only for the sake of this example, so that it's true.

If we have control over the implementation of `Document` and `Folder` we can implement this feature easily by adding a `size` method to both classes:

```
class Document(object):
    def __init__(self, text):
        self.text = text

    def size(self):
```

```
        return len(self.text)

class Folder(object):
    def __init__(self, items):
        self.items = items

    def size(self):
        return sum([item.size() for item in self.items])
```

And then we can simply call the `.size()` method to get the size:

```
>>> doc = Document('Hello world!')
>>> doc.size()
12
>>> doc2 = Document('Bye world!')
>>> doc2.size()
10
>>> folder = Folder([doc, doc2])
>>> folder.size()
22
```

Note that the `Folder` size code is generic; it doesn't care what the items inside it are; if they have a `size` method that gives the right result, it will work. If a new content item `Image` is defined and we provide a `size` method for this, a `Folder` instance that contains `Image` instances will still be able to calculate its size. Let's try this:

```
class Image(object):
    def __init__(self, bytes):
        self.bytes = bytes

    def size(self):
        return len(self.bytes)
```

When we add an `Image` instance to the folder, the size of the folder can still be calculated:

```
>>> image = Image('abc')
>>> folder.items.append(image)
>>> folder.size()
25
```

2.3 Adding `size` from outside

Open/Closed Principle

The Open/Closed principle states software entities should be open for extension, but closed for modification. The idea is you may have a piece of software that you cannot or do not want to change, for instance because it's being developed by a third party, or because the feature you want to add is outside of the scope of that software (separation of concerns). By extending the software without modifying its source code, you can benefit from the stability of the core software and still add new functionality.

So far we didn't need Reg at all. But in the real world things may be a lot more complicated. We may be dealing with a content management system core where we *cannot* control the implementation of `Document` and `Folder`. What if we want to add a size calculation feature in an extension package?

We can fall back on good-old Python functions instead. We separate out the size logic from our classes:

```
def document_size(document):
    return len(document.text)

def folder_size(folder):
    return sum([document_size(item) for item in folder.items])
```

2.4 Generic size

What about monkey patching?

We *could* monkey patch a `size` method into all our content classes. This would work. But doing this can be risky – what if the original CMS's implementers change it so it *does* gain a size method or attribute, for instance? Multiple monkey patches interacting can also lead to trouble. In addition, monkey-patched classes become harder to read: where is this `size` method coming from? It isn't there in the `class` statement, or in any of its superclasses! And how would we document such a construction?
In short, monkey patching does not make for very maintainable code.

There is a problem with the above implementation however: `folder_size` is not generic anymore, but now depends on `document_size`. It would fail when presented with a folder with an `Image` in it:

```
>>> folder_size(folder)
Traceback (most recent call last):
  ...
AttributeError: ...
```

To support `Image` we first need an `image_size` function:

```
def image_size(image):
   return len(image.bytes)
```

We can now write a generic `size` function to get the size for any item we give it:

```
def size(item):
    if isinstance(item, Document):
        return document_size(item)
    elif isinstance(item, Image):
        return image_size(item)
    elif isinstance(item, Folder):
        return folder_size(item)
    assert False, "Unknown item: %s" % item
```

With this, we can rewrite `folder_size` to use the generic `size`:

```
def folder_size(folder):
    return sum([size(item) for item in folder.items])
```

Now our generic `size` function will work:

```
>>> size(doc)
12
>>> size(image)
3
>>> size(folder)
25
```

All a bit complicated and hard-coded, but it works!

2.5 New `File` content

What if we now want to write a new extension to our CMS that adds a new kind of folder item, the `File`, with a `file_size` function?

```
class File(object):
   def __init__(self, bytes):
       self.bytes = bytes

def file_size(file):
    return len(file.bytes)
```

We would need to remember to adjust the generic `size` function so we can teach it about `file_size` as well. Annoying, tightly coupled, but sometimes doable.

But what if we are actually yet another party, and we have control of neither the basic CMS *nor* its size extension? We cannot adjust `generic_size` to teach it about `File` now! Uh oh!

Perhaps the implementers of the size extension were wise and anticipated this use case. They could have implemented `size` like this:

```
size_function_registry = {
   Document: document_size,
   Image: image_size,
   Folder: folder_size
}

def register_size(class_, function):
   size_function_registry[class_] = function

def size(item):
   return size_function_registry[item.__class__](item)
```

We can now use `register_size` to teach `size` how to get the size of a `File` instance:

```
register_size(File, file_size)
```

And it would work:

```
>>> size(File('xyz'))
3
```

This is quite a bit of custom work on the parts of the implementers, though. The API to manipulate the size registry is also completely custom. But you can do it.

2.6 New `HtmlDocument` content

What if we introduce a new `HtmlDocument` item that is a subclass of `Document`?

```
class HtmlDocument(Document):
    pass # imagine new html functionality here
```

Let's try to get its size:

```
>>> htmldoc = HtmlDocument('<p>Hello world!</p>')
>>> size(htmldoc)
Traceback (most recent call last):
```

```
    ...
KeyError: ...
```

Uh oh, that doesn't work! There's nothing registered for the `HtmlDocument` class.

We need to remember to also call `register_size` for `HtmlDocument`. We can reuse `document_size`:

```
>>> register_size(HtmlDocument, document_size)
```

Now `size` will work:

```
>>> size(htmldoc)
19
```

This is getting rather complicated, requiring not only foresight and extra implementation work for the developers of `size` but also extra work for the person who wants to subclass a content item.

Hey, we should write a system that generalizes this and automates a lot of this, and gives us a more universal registry API, making our life easier! And that's Reg.

2.7 Doing this with Reg

Let's see how we could implement `size` using Reg.

First we need our generic `size` function:

```
def size(obj):
    raise NotImplementedError
```

This function raises `NotImplementedError` as we don't know how to get the size for an arbitrary Python object. Not very useful yet. We need to be able to hook the actual implementations into it. To do this, we first need to transform the `size` function to a generic one:

```
import reg
size = reg.generic(size)
```

We can actually spell these two steps in a single step, as `reg.generic()` can be used as decorator:

```
@reg.generic
def size(obj):
    raise NotImplementedError
```

We can now register the various size functions for the various content items in a registry:

```
r = reg.Registry()
r.register(size, [Document], document_size)
r.register(size, [Folder], folder_size)
r.register(size, [Image], image_size)
r.register(size, [File], file_size)
```

We can now use our `size` function:

```
>>> size(doc, lookup=r)
12
```

The `lookup` argument

What's this `lookup` argument about? It lets you specify explicitly what registry Reg looks in to look up the size functions, on our case `r`.
If we forget it, we'll get an error:

```
>>> size(doc)
Traceback (most recent call last):
  ...
NoImplicitLookupError: Cannot lookup without explicit lookup argument because no implici lookup was
```

If your generic function implementation defines a `lookup` argument it will receive the lookup used. This way you can continue passing the lookup along explicitly from generic function to generic function if you want to.
It's annoying to have to keep spelling this out all the time – we don't do it in our `folder_size` implementation, for instance, so that will fail too, even if we pass a lookup to the our `size` function, as it won't be passed along implicitly.

```
>>> size(folder, lookup=r)
Traceback (most recent call last):
  ...
NoImplicitLookupError: Cannot lookup without explicit lookup argument because no implici lookup was
```

Using `reg.implicit.Implicit.initialize()` we can specify an implicit lookup argument for all generic lookups so we don't have to pass it in anymore:

```
from reg import implicit
implicit.initialize(r)
```

Now we can just call our new generic `size`:

```
>>> size(doc)
12
```

And it will work for folder too:

```
>>> size(folder)
25
```

It will work for subclasses too:

```
>>> size(htmldoc)
19
```

Reg knows that `HtmlDocument` is a subclass of `Document` and will find `document_size` automatically for you. We only have to register something for `HtmlDocument` if we would want to use a special, different size function for `HtmlDocument`.

2.8 Using classes

The previous example worked well for a single function to get the size, but what if we wanted to add a feature that required multiple methods, not just one?

Let's imagine we have a feature to get the icon for a content object in our CMS, and that this consists of two methods, with a way to get a small icon and a large icon. We want this API:

```
from abc import ABCMeta, abstractmethod
```

```
class Icon(object):
    __metaclass__ = ABCMeta
    @abstractmethod
    def small(self):
        """Get the small icon."""

    @abstractmethod
    def large(self):
        """Get the large icon."""
```

abc module?

We've used the standard Python abc module to set the API in stone. But that's just a convenient standard way to express it. The `abc` module is not in any way required by Reg. You don't need to implement the API in a base class either. We just do it in this example to be explicit.

Let's implement the `Icon` API for `Document`:

```
def load_icon(path):
    return path # pretend we load the path here and return an image obj

class DocumentIcon(Icon):
   def __init__(self, document):
      self.document = document

   def small(self):
      if not self.document.text:
          return load_icon('document_small_empty.png')
      return load_icon('document_small.png')

   def large(self):
      if not self.document.text:
          return load_icon('document_large_empty.png')
      return load_icon('document_large.png')
```

The constructor of `DocumentIcon` receives a `Document` instance as its first argument. The implementation of the `small` and `large` methods uses this instance to determine what icon to produce depending on whether the document is empty or not.

We can call `DocumentIcon` an adapter, as it adapts the original `Document` class to provide an icon API for it. We can use it manually:

```
>>> icon_api = DocumentIcon(doc)
>>> icon_api.small()
'document_small.png'
>>> icon_api.large()
'document_large.png'
```

But we want to be able to use the `Icon` API in a generic way, so let's create a generic function that gives us an implementation of `Icon` back for any object:

```
@reg.generic
def icon(obj):
    raise NotImplementedError
```

We can now register the `DocumentIcon` adapter class for this function and `Document`:

```
r.register(icon, [Document], DocumentIcon)
```

We can now use the generic `icon` to get `Icon` API for a document:

```
>>> api = icon(doc)
>>> api.small()
'document_small.png'
>>> api.large()
'document_large.png'
```

We can also register a `FolderIcon` adapter for `Folder`, a `ImageIcon` adapter for `Image`, and so on. For the sake of brevity let's just define one for `Image` here:

```
class ImageIcon(Icon):
    def __init__(self, image):
        self.image = image

    def small(self):
        return load_icon('image_small.png')

    def large(self):
        return load_icon('image_large.png')

r.register(icon, [Image], ImageIcon)
```

Now we can use `icon` to retrieve the `Icon` API for any item in the system for which an adapter was registered:

```
>>> icon(doc).small()
'document_small.png'
>>> icon(doc).large()
'document_large.png'
>>> icon(image).small()
'image_small.png'
>>> icon(image).large()
'image_large.png'
```

2.9 Multiple dispatch

Sometimes we want to adapt more than one thing at the time. The canonical example for this is a web view lookup system. Given a request and a model, we want to find a view that represents these. The view needs to get the request, for parameter information, POST body, URL information, and so on. The view also needs to get the model, as that is what will be represented in the view.

You want to be able to vary the view depending on the type of the request as well as the type of the model.

Let's imagine we have a `Request` class:

```
class Request(object):
    pass
```

We'll use `Document` as the model class.

We want a generic `view` function that given a request and a model generates content for it:

```
@reg.generic
def view(request, model):
    raise NotImplementedError
```

We now define a concrete view for `Document`:

```
def document_view(request, document):
    return "The document content is: " + document.text
```

Let's register the view in the registry:

```
r.register(view, [Request, Document], document_view)
```

We now see why the second argument to `register()` is a list; so far we only supplied a single entry in it, but here we supply two, as we have two parameters on which to do dynamic dispatch.

Given a request and a document, we can now adapt it to `IView`:

```
>>> request = Request()
>>> view(request, doc)
'The document content is: Hello world!'
```

CHAPTER 3

Service Discovery

Sometimes you want your application to have configurable services. The application may for instance need a way to send email, but you don't want to hardcode any particular way into your app, but instead leave this to a particular deployment-specific configuration. You can use the Reg infrastructure for this as well.

The simplest way to do this with Reg is by using a generic service lookup function:

```
@reg.generic
def emailer():
    raise NotImplementedError
```

Here we've created a generic function that takes no arguments (and thus does no dynamic dispatch). But it's still generic, so we can plug in its actual implementation elsewhere, into the registry:

```
sent = []

def send_email(sender, subject, body):
    # some specific way to send email
    sent.append((sender, subject, body))

def actual_emailer():
    return send_email

r.register(emailer, [], actual_emailer)
```

Now when we call emailer, we'll get the specific service we want:

```
>>> the_emailer = emailer()
>>> the_emailer('someone@example.com', 'Hello', 'hello world!')
>>> sent
[('someone@example.com', 'Hello', 'hello world!')]
```

In this case we return the function `send_email` from the `emailer()` function, but we could return any object we want that implements the service, such as an instance with a more extensive API.

CHAPTER 4

Lower level API

4.1 Registering non-functions

Some special use cases require the registration of other objects besides callables. Reg exposes an API to get at these:

```
@reg.generic
def foo(model):
    raise NotImplementedError

thing = "Thing"

r.register(foo, [Document], thing)
```

We've registered `thing` for generic `foo` of `Document` now, not a function. Because `thing` is not a function, calling `foo` for `Document` will result in an error:

```
>>> foo(doc)
Traceback (most recent call last):
  ...
TypeError: 'str' object is not callable
```

We can still get at `thing` with a special method on the function called `component`:

```
>>> foo.component(doc)
"thing"
```

4.2 Getting all

As we've seen, Reg supports inheritance. `size` for instance was registered for `Document` instances, and is therefore also available of instances of its subclass, `HtmlDocument`:

```
>>> size.component(doc) is document_size
True
>>> size.component(htmldoc) is document_size
True
```

Using the special `all` function we can also get an iterable of *all* the components registered for a particular instance, including those of base classes. Right now this is pretty boring as there's only one of them:

```
>>> list(size.all(doc))
[<function document_size at ...>]
```

```
>>> list(size.all(htmldoc))
[<function document_size at ...>]
```

We can make this more interesting by registering a special `htmldocument_size` to handle `HtmlDocument` instances:

```
def htmldocument_size(doc):
   return len(doc.text) + 1 # 1 so we can see a difference

r.register(size, [HtmlDocument], htmldocument_size)
```

`size.all()` for `htmldoc` now also gives back the more specific `htmldocument_size`:

```
>>> list(size.all(htmldoc))
[<function htmldocument_size at ...>, <function document_size at ...>]
```

4.3 Using the Registry directly

The key under which we register something in a registry in fact doesn't need to be a function. We can use any hashable object, such as a string:

```
r.register('some key', [Document], 'some registered')
```

We can't get it at it using a generic dispatch function anymore now. We can use the `reg.Lookup` API instead (in this case it's provided by `Registry` directly). Here's what to do:

```
>>> r.component('some key', [doc])
'some registered'
>>> list(r.all('some key', [doc]))
['some registered']
```

CHAPTER 5

Composition

Reg separates the registration API from the lookup API. The `Registry` implementation we've been using combines both in one, but we can separate the two instead. This is useful for a framework developer that may want to allow the composition of multiple lookups together. It also supports caching lookups to help performance.

5.1 ClassRegistry

`reg.ClassRegistry` does not offer the full lookup API but does still allows registration:

```
cr = reg.ClassRegistry()
```

We can use this to do registration as before:

```
@reg.generic
def example():
    raise NotImplementedError

def document_example(doc):
    return "Document Example"

cr.register(example, [Document], document_example)
```

So far nothing is different. But `ClassRegistry` supports the *class lookup* API that lets you lookup registrations by the *class* of what was registered instead of by instance. Here's how:

```
>>> cr.get(example, [Document])
<function document_example at ...>
```

It is still inheritance aware, too:

```
>>> cr.get(example, [HtmlDocument])
<function document_example at ...>
```

We can get the original instance-based lookup API from a class lookup by wrapping it in a `Lookup`:

```
>>> l = reg.Lookup(cr)
>>> l.component(example, [doc])
<function document_example at ...>
```

5.2 Caching

Now the fun starts. We can turn a class lookup in a faster, caching class lookup using `reg.CachingClassLookup`:

```
>>> caching = reg.CachingClassLookup(cr)
>>> caching.get(example, [Document])
<function document_example at ...>
```

Turning it back into a lookup gives us a caching version of what we had before:

```
>>> caching_lookup = reg.Lookup(caching)
>>> caching_lookup.component(example, [doc])
<function document_example at ...>
```

You'll have to trust us on this, but it's faster the second time as it's cached!

5.3 Composing class lookups

You can also compose class lookups together into a bigger class lookup. This allows you to compose and partition behavior, sharing behavior where you want it but isolating it otherwise.

The use case for this is a core framework that provides default behavior, with applications written on top that extend or override this default behavior. If one application overrides the behavior, another application written on top of the same framework should not be affected.

Let's look at an example of this. First we define three registries: for the framework, for one application built with it, and for another application built with it:

```
framework = reg.ClassRegistry()
app = reg.ClassRegistry()
other_app = reg.ClassRegistry()
```

We can now compose the `framework` and the `app` class lookup using `reg.ListClassLookup`:

```
app_combined = reg.Lookup(reg.ListClassLookup([app, framework]))
```

We compose the `framework` and the `other_app` class lookup separately:

```
other_app_combined = reg.Lookup(reg.ListClassLookup([other_app, framework]))
```

Our hypothetical example framework provides a serialization API. The idea is that we can call `serialize` on an object to get a representation of that object as dictionaries and lists, JSON-style:

```
@reg.generic
def serialize(obj):
    raise NotImplementedError
```

We've also provided a default serialization for documents in our framework:

```
def document_serialize(doc):
    return { 'text': doc.text }

framework.register(serialize, [Document], document_serialize)
```

Let's try it with the core framework itself:

```
>>> serialize(doc, lookup=reg.Lookup(framework))
{'text': 'Hello world!'}
```

It also works in the `app_combined` application and the `other_app_combined` application:

```
>>> serialize(doc, lookup=app_combined)
{'text': 'Hello world!'}
>>> serialize(doc, lookup=other_app_combined)
{'text': 'Hello world!'}
```

Now we decide that we want to override the default serialization for `Document`, but only in `app`, not in the framework itself, so that `other_app` is unaffected:

```
def app_document_serialize(doc):
   return { 'content': 'The content: %s' % doc.text }

app.register(serialize, [Document], app_document_serialize)
```

Our application has the new behavior now:

```
>>> serialize(doc, lookup=app_combined)
{'content': 'The content: Hello world!'}
```

But our framework is not affected, and neither is `other_app`:

```
>>> serialize(doc, lookup=reg.Lookup(framework))
{'text': 'Hello world!'}
>>> serialize(doc, lookup=other_app_combined)
{'text': 'Hello world!'}
```

So far in this example we've used the explicit `lookup` argument. But how does this combine with the implict lookup facility? Changing the implicit lookup before each application switch seems daunting, but in practice you'd typically only switch the implicit application context once per thread. The implicit lookup is thread local, so that one thread's implicit lookup does not affect the other. Multiple threads can this way run different applications all sharing the same framework. This does require doing all the required registrations during application startup time, and then not modifying them anymore during run time, as registration is not thread-safe, just lookup.

CHAPTER 6

API

class reg.**IRegistry**

A registration API for components.

clear()

Clear registry of all registrations.

exact(*key*, *classes*)

Get registered component for exactly key and classes.

Parameters

- **key** (*hashable object, normally function.*) – Get component for this key.
- **classes** (*list of classes.*) – List of classes for which to get component.

Returns registered component, or None.

Does not go to base classes, just returns exact registration.

Returns None if no registration exists.

register(*key*, *classes*, *component*)

Register a component.

Parameters

- **key** (*hashable object, normally function.*) – Register component for this key.
- **classes** (*list of classes.*) – List of classes for which to register component.
- **component** (*object.*) – Any python object, often a function. Can be a reg.Matcher instance.

The key is a hashable object, often a function object, by which the component can be looked up.

classes is a list of 0 to n classes that the component is registered for. If multiple sources are listed, a registration is made for that combination of sources.

The component is a python object (function, class, instance, etc) that is registered. If you're working with multiple dispatch, you would register a function that expects instances of the classes in classes as its arguments.

class reg.**IClassLookup**

all(*key*, *classes*)

Look up all components, by key and classes.

Parameters

- **key** (*hashable object, normally function.*) – Get components for this key.
- **classes** (*list of classes.*) – List of classes for which to get components.

Returns iterable of found components.

The key is a hashable object, often a function object, by which the components are looked up.

classes is a list of 0 to n classes that we use to look up the components. If multiple classes are listed, the lookup is made for that combination of classes. All registered components for combinations of base classes are also returned.

A Cartesian product is made of all combinations of base classes to do this, sorted by inheritance, first class to last class, most specific to least specific.

This calculation is relatively expensive so you can wrap a class lookup in a `reg.CachingClassLookup` proxy to speed up subsequent calls.

If no components can be found, the iterable returned will be empty.

get (*key*, *classes*)
Look up a component, by key and classes of arguments.

Parameters

- **key** (*hashable object, normally function.*) – Get component for this key.
- **classes** (*list of classes.*) – List of classes for which to get component.

Returns registered component, or `None`.

The key is a hashable object, often a function object, by which the component is looked up.

classes is a list of 0 to n classes that we use to look up the component. If multiple classes are listed, the lookup is made for that combination of classes.

In order to find the most matching registered component, a Cartesian product is made of all combinations of base classes given, sorted by inheritance, first class to last class, most specific to least specific.

This calculation is relatively expensive so you can wrap a class lookup in a `reg.CachingClassLookup` proxy to speed up subsequent calls.

If the component can be found, it will be returned. If the component cannot be found, `None` is returned.

class `reg.`**ClassRegistry**
Bases: `reg.registry.IRegistry`, `reg.registry.IClassLookup`

class `reg.`**Registry**
Bases: `reg.registry.IRegistry`, `reg.lookup.Lookup`

class `reg.`**Lookup** (*class_lookup*)
Look up objects for a key.

The lookup API is also available directly on a function decorated with `reg.generic()`. The `call` method stands in for the actual function call. If the call method is in use from `reg.generic`, `ComponentLookupError` is never raised, and instead the fall back is to the function being decorated.

all (*key*, *args*, *predicates=None*)
Lookup up all components registered for args.

Parameters

- **key** (*hashable object, normally function.*) – Look up components for this key.
- **args** (*list of objects.*) – Look up components for these arguments.
- **predicates** (*dict.*) – predicates (used by Matcher)

Returns iterable of registered components.

The behavior of this method is like that of component, but it looks up *all* the matching components for the arguments. This means that if one component is registered for a class and another for its base class, `all()` with an instance of the class as its argument will return both components.

Will check whether the found component is an Matcher, in which case it will be called with args. If non-None is returned, the found value is included as a matching component. If a matcher is involved and the `predicates` parameter is supplied, this will be used for the matcher, overriding any predicate calculation it may do itself. Otherwise the `predicates` parameter has no effect.

If no components can be found, the iterable will be empty.

call(*key*, *args*, *default=<Sentinel>*, ***kw*)

Call function based on multiple dispatch on args.

Parameters

- **key** (*hashable object, normally function.*) – Call function for this key.
- **args** (*list of objects.*) – Call function with these arguments.
- **default** (*object.*) – default value to return if lookup fails.
- **kw** – extra keyword arguments passed to the function called.

Returns result of function call.

Raises ComponentLookupError

The behavior of this method is like that of component, but it performs an extra step: it calls the found component with the args given as arguments.

This amounts to an implementation of multiple dispatch: zero or more arguments can be used to dispatch the function on.

If the found component has a lookup argument, it will pass the lookup to this argument too. This allows you to pass along lookup completely explicitly between generic functions.

component(*key*, *args*, *default=<Sentinel>*, *predicates=None*)

Look up a component.

Parameters

- **key** (*hashable object, normally function.*) – Look up component for this key.
- **args** (*list of objects.*) – Look up component for these arguments.
- **default** (*object.*) – default value to return if lookup fails.
- **predicates** (*dict.*) – optional predicate ditcionary for matcher, overriding the matcher's predicate calculation.

Returns registered component.

Raises ComponentLookupError

A component can be any Python object.

key is a hashable object that is used to determine what to look up. Normally it is a Python function.

args is a list of 0 to n objects that we use to look up the component. The classes of the args are used to do the look up. If multiple args are listed, the lookup is made for that combination of args.

If the component found is an instance of class:*Matcher*, it will be called with args as parameters (`matcher(*args)`). The matcher can return an object, in which case will be returned as the real matching component. If the matcher returns `None` it will look for a match higher up the ancestor chain of args.

If a `predicates` argument is supplied this is used by the matcher instead of doing its own predicate calculation from the arguments. This can be useful in combination with the `reg.PredicateMatcher` to override which predicates are used in a lookup.

If a component can be found, it will be returned. If the component cannot be found, a `ComponentLookupError` will be raised, unless a default argument is specified, in which case it will be returned.

class `reg.`**`Matcher`**
Look up by calling and returning value.

If a component that subclasses Matcher is registered, it it is called with args, i.e. `matcher(*args)`. The resulting value is considered to be the looked up component. If the resulting value is `None`, no component is found for this matcher.

A matcher can be found multiple times during a lookup (if the first matcher results in `None`. Information such as predicates may have to be calculated multiple times in that case. This can be avoided by defining a `predicates` method which takes the arguments used for the lookup as arguments. The result should be a dictionary which is passed as keyword arguments into this matcher, as well as any further candidate matchers if this one returns `None`.

exception `reg.`**`ComponentLookupError`**
Error raised when a component cannot be found.

Will only be raised if nod `default` argument was supplied during lookup.

class `reg.`**`ListClassLookup`**(*lookups*)
Bases: `reg.registry.IClassLookup`

A simple list of class lookups functioning as a single IClassLookup.

Go through all items in the list, starting at the beginning and try to find the component. If found in a lookup, return it right away.

class `reg.`**`ChainClassLookup`**(*lookup*, *next*)
Bases: `reg.registry.IClassLookup`

Chain a class lookup on top of another class lookup.

Look in the supplied IClassLookup object first, and if not found, look in the next IClassLookup object. This way multiple IClassLookup objects can be chained together.

class `reg.`**`CachingClassLookup`**(*class_lookup*)
Bases: `reg.registry.IClassLookup`

Cache an existing class lookup.

All previous accesses to class lookup are cached.

`reg.`**`generic`**(*func*)
Turn a function into a generic function.

Parameters **func** (*function.*) – Function to turn into a multiple dispatch function.

Returns multiple dispatch version of function.

When someone calls the wrapped function, the arguments determine what actual function will be called. In particular the classes of the arguments are inspected. For each combination of arguments a different function can be registered.

The function itself provides a default implementation in case no more specific registered function can be found for its arguments.

Can be used as a decorator:

```
@reg.generic
def my_function(...):
    ...
```

class reg.**PredicateRegistry**(*predicates*)
A registry that can be used to index items by predicate.

class reg.**Predicate**(*name*, *index_factory*, *calc=None*, *default=None*)
A predicate.

class reg.**KeyIndex**
An index for matching predicates by key.

exception reg.**PredicateRegistryError**
An error using the predicate registry.

class reg.implicit.**Implicit**
Implicit global lookup.

There will only one singleton instance of this, called `reg.implicit`. The lookup can then be accessed using `reg.implicit.lookup`.

Generic functions as well as their `component` and `all` methods make use of this information if you do not pass an explicit `lookup` argument to them. This is handy as it becomes unnecessary to have to pass a `lookup` object everywhere.

The drawback is that this single global lookup is implicit, which makes it harder to test in isolation. Reg supports testing with the explicit `lookup` argument, but that is not useful if you are testing code that relies on an implicit lookup. Therefore Reg strives to make the implicit global lookup as explicit as possible so that it can be manipulated in tests where this is necessary.

It is also possible for a framework to change the implicit lookup during run-time. This is done by simply assigning to `implicit.lookup`. The lookup is stored on a thread-local and is unique per thread.

Reg offers facilities to compose such a custom lookup:

- `reg.ListClassLookup` and `reg.ChainClassLookup` which can be used to chain multiple `IClassLookup` instances together.
- `reg.CachingClassLookup` which can be used to create a faster caching version of an `IClassLookup`.
- `reg.Lookup` which can be used to turn a `IClassLookup` into a proper `ILookup`.

To change the lookup back to a lookup in the global implicit registry, call `reset`.

The implicit lookup is thread-local: each thread has a separate implicit global lookup.

clear()
Clear global implicit lookup.

initialize(*lookup*)
Initialize implicit with lookup.

Parameters **lookup** (*ILookup.*) – The lookup that will be the global implicit lookup.

lookup
Get the implicit ILookup.

reset()
Reset global implicit lookup to original lookup.

This can be used to wipe out any composed lookups that were installed in this thread.

exception `reg.`**`NoImplicitLookupError`**
No implicit lookup was registered.

Register an implicit lookup by calling *reg.implicit.initialize()*, or pass an explicit `lookup` argument to generic function calls.

`reg.`**`mapply`**(*func*, **args*, ***kw*)
Apply keyword arguments to function only if it defines them.

So this works without error as `b` is ignored:

```
def foo(a):
    pass

mapply(foo, a=1, b=2)
```

Zope has an mapply that does this but a lot more too. py.test has an implementation of getting the argument names for a function/method that we've borrowed.

`reg.`**`arginfo`**(*callable*)
Get information about the arguments of a callable.

Returns a `inspect.ArgSpec` object as for `inspect.getargspec()`.

`inspect.getargspec()` returns information about the arguments of a function. arginfo also works for classes and instances with a __call__ defined. Unlike getargspec, arginfo treats bound methods like functions, so that the self argument is not reported.

arginfo caches previous calls (except for instances with a __call__), making calling it repeatedly cheap.

This was originally inspired by the pytest.core varnames() function, but has been completely rewritten to handle class constructors, also show other getarginfo() information, and for readability.

CHAPTER 7

Developing Reg

7.1 Install Reg for development

First make sure you have virtualenv installed for Python 2.7.

Now create a new virtualenv somewhere for Reg's development:

```
$ virtualenv /path/to/ve_reg
```

The goal of this is to isolate you from any globally installed versions of setuptools, which may be incompatible with the requirements of the buildout tool. You should also be able to recycle an existing virtualenv, but this method guarantees a clean one.

Clone Reg from github (https://github.com/morepath/reg) and go to the reg directory:

```
$ git clone git@github.com:morepath/reg.git
$ cd reg
```

Now we need to run bootstrap.py to set up buildout, using the Python from the virtualenv we've created before:

```
$ /path/to/ve_reg/bin/python bootstrap.py
```

This installs buildout, which can now set up the rest of the development environment:

```
$ bin/buildout
```

This will download and install various dependencies and tools.

7.2 Running the tests

You can run the tests using py.test. Buildout will have installed it for you in the `bin` subdirectory of your project:

```
$ bin/py.test reg
```

To generate test coverage information as HTML do:

```
$ bin/py.test --cov reg --cov-report html
```

You can then point your web browser to the `htmlcov/index.html` file in the project directory and click on modules to see detailed coverage information.

7.3 Running the documentation tests

The documentation contains code. To check these code snippets, you can run this code using this command:

```
$ bin/sphinx-build  -b doctest doc out
```

7.4 Building the HTML documentation

To build the HTML documentation (output in `doc/build/html`), run:

```
$ bin/sphinxbuilder
```

7.5 Various checking tools

The buildout will also have installed flake8, which is a tool that can do various checks for common Python mistakes using pyflakes, check for PEP8 style compliance and can do cyclomatic complexity checking. To do pyflakes and pep8 checking do:

```
$ bin/flake8 reg
```

To also show cyclomatic complexity, use this command:

```
$ bin/flake8 --max-complexity=10 reg
```

CHAPTER 8

History of Reg

Reg was written by me, Martijn Faassen; the core mapping code was originally co-authored by Thomas Lotze.

Reg is heavily inspired by the Zope Component Architecture (ZCA), namely the `zope.interface` and `zope.component` packages. Reg is however a completely different codebase with an entirely different API. At the end I've included a brief history of the ZCA.

8.1 Reg History

The Reg code went through a bit of history:

The core registry (mapping) code was conceived by Thomas Lotze and myself as a speculative sandbox project in January of 2010. It was called `iface` then:

http://svn.zope.org/Sandbox/faassen/iface/

In early 2012, I was at a sprint in Nürnberg, Germany organized by Novareto. Inspired by discussions with the sprint participants, particularly the Cromlech developers Souheil Chelfouh and Alex Garel, I created a project called Crom:

https://github.com/faassen/crom

Crom focused on rethinking component and adapter registration and lookup APIs, but was still based on `zope.interface` for its fundamental `AdapterRegistry` implementation and the `Interface` metaclass. I worked a bit on Crom after the sprint, but soon I moved on to other matters.

At the Plone conference held in Arnhem, the Netherlands in October 2012, I gave a lightning talk about Crom. I figured what Crom needed was a rewrite of the core adapter registry, i.e. what was in the iface project. In the end of 2012 I mailed Thomas Lotze asking whether I could merge iface into Crom, and he gave his kind permission.

The core registry code of iface was never quite finished however, and while the iface code was now in Crom, Crom didn't use it yet. Thus it lingered some more.

In July 2013 in development work for CONTACT (contact.de), I found myself in need of clever registries. Crom also had some configuration code intermingled with the component architecture code and I didn't want this anymore.

So I reorganized the code yet again into another project, this one: Reg. I then finished the core mapping code and hooked it up to the Crom-style API, which I refactored further. For interfaces, I used Python's `abc` module.

For a while during internal development this codebase was called `Comparch`, but this conflicted with another name so I decided to call it `Reg`, short for registry, as it's really about clever registries more than anything else.

After my first announcement of Reg to the world in september 2013 I got the question why I shouldn't just use PEP 443, which has a generic function implementation (single dispatch). I started thinking I should convert Reg to a generic function implementation, as it was already very close. After talking to some people about this at PyCon DE in october,

I did the refactoring to use generic functions throughout and no interfaces for lookup, and this is the current Reg you see.

8.2 Brief history of Zope Component Architecture

Reg is heavily inspired by `zope.interface` and `zope.component`, by Jim Fulton and a lot of Zope developers. `zope.interface` has a long history, going all the way back to December 1998, when a scarecrow interface package was released for discussion:

http://old.zope.org/Members/jim/PythonInterfaces/Summary/

http://old.zope.org/Members/jim/PythonInterfaces/Interface/

A later version of this codebase found itself in Zope, as `Interface`:

http://svn.zope.org/Zope/tags/2-8-6/lib/python/Interface/

A new version called zope.interface was developed for the Zope 3 project, somewhere around the year 2001 or 2002 (code historians, please dig deeper and let me know). On top of this a zope.component library was constructed which added registration and lookup APIs on top of the core zope.interface code.

zope.interface and zope.component are widely used as the core of the Zope 3 project. zope.interface was adopted by other projects, such as Zope 2, Twisted, Grok, BlueBream and Pyramid.

CHAPTER 9

Indices and tables

- *genindex*
- *modindex*
- *search*

r

reg, 23