Reg Documentation

Release 0.10

Martijn Faassen

October 04, 2016

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 implements *predicate dispatch* and *multiple registries*:

Predicate dispatch

> We all know about dynamic dispatch: when you call a method on an instance it is dispatched to the implementation in its class, and the class is determined from the first argument (`self`). This is known as *single dispatch*.
>
> Reg implements multiple dispatch. This is a generalization of single dispatch: multiple dispatch allows you to dispatch on the class of *other* arguments besides the first one.
>
> Reg actually implements predicate dispatch, which is a further generalization that allows dispatch on *arbitrary properties* of arguments, instead of just their class.
>
> The Morepath web framework is built with Reg. It uses Reg's predicate dispatch system. Its full power can be seen in its view lookup system.
>
> This document explains how to use Reg. Various specific patterns are documented in Patterns.

Multiple registries

> Reg supports an advanced application architecture pattern where you have multiple predicate dispatch registries in the same runtime. This means that dispatch can behave differently depending on runtime context. You do this by using dispatch *methods* that you associate with a class that represents the application context. When you switch the context class, you switch the behavior.
>
> Morepath uses context-based dispatch to support its application composition system, where one application can be mounted into another.
>
> See Context-based dispatch for this advanced application pattern.

Reg is designed with a caching layer that allows it to support these features efficiently.

1.2 Example

Let's examine a short example. First we use the *`reg.dispatch()`* decorator to define a function that dispatches based on the class of its `obj` argument:

```
import reg

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

We want this function to return the title of its `obj` argument.

Now we create a few example classes for which we want to be able to use the `title` fuction we defined above.

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

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

If we call `title` with a `TitledReport` instance, we want it to return its `title` attribute:

```
@title.register(obj=TitledReport)
def titled_report_title(obj):
    return obj.title
```

The `title.register` decorator registers the function `titled_report_title` as an implementation of `title` when `obj` is an instance of `TitleReport`.

There is also a more programmatic way to register implementations. Take for example, the implementation of `title` with a `LabeledReport` instance, where we want it to return its `label` attribute:

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

We can register it by explicitely invoking `title.register`:

```
title.register(labeled_report_title, obj=LabeledReport)
```

Now the generic `title` function works on both titled and labeled objects:

```
>>> titled = TitledReport('This is a report')
>>> labeled = LabeledReport('This is also a report')
>>> title(titled)
'This is a report'
>>> title(labeled)
'This is also a report'
```

What is going on and why is this useful at all? We present a worked out example next.

1.3 Dispatch functions

1.3.1 A Hypothetical CMS

Let's look at how Reg works in 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 entries, 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, entries):
       self.entries = entries
```

1.3.2 `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 for the sake of this example.

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, entries):
       self.entries = entries

   def size(self):
       return sum([entry.size() for entry in self.entries])
```

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
```

The `Folder` size code is generic; it doesn't care what the entries 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.entries.append(image)
>>> folder.size()
25
```

Cool! So we're done, right?

1.3.3 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 that 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 a real world CMS we aren't always in the position to change the content classes themselves. We may be dealing with a content management system core where we *cannot* control the implementation of `Document` and `Folder`. Or perhaps we can, but we want to keep our code modular, in independent packages. So how would we 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(item):
    return len(item.text)

def folder_size(item):
    return sum([document_size(entry) for entry in item.entries])
```

1.3.4 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 function-based implementation however: `folder_size` is not generic anymore, but now depends on `document_size`. It fails 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(item):
   return len(item.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(item):
    return sum([size(entry) for entry in item.entries])
```

Now our generic `size` function works:

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

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

1.3.5 New `File` content

What if we 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(item):
    return len(item.bytes)
```

We 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 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 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 works:

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

But this is quite a bit of custom work that the implementers need to do, and it involves a new API (`register_size`) to manipulate the `size_function_registry`. But it can be done.

1.3.6 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: ...
```

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 automates a lot of this, and gives us a universal registration API, making our life easier! And what if we want to switch behavior based on more than just one argument? Maybe you even want different dispatch behavior depending on application context? This is what Reg is for.

1.3.7 Doing this with Reg

Let's see how we can implement `size` using Reg:

First we need our generic `size` function:

```
def size(item):
    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.dispatch('item')(size)
```

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

```
@reg.dispatch('item')
def size(item):
    raise NotImplementedError
```

What this says that when we call `size`, we want to dispatch based on the class of its `item` argument.

We can now register the various size functions for the various content items as implementations of `size`:

```
size.register(document_size, item=Document)
size.register(folder_size, item=Folder)
size.register(image_size, item=Image)
size.register(file_size, item=File)
```

`size` now works:

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

It works for folder too:

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

It works 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 want to use a special, different size function for `HtmlDocument`.

1.4 Multiple and predicate dispatch

Let's look at an example where dispatching on multiple arguments is useful: a web view lookup system. Given a request object that represents a HTTP request, and a model instance (document, icon, etc), we want to find a view function that knows how to make a representation of the model given the request. Information in the request can influence the representation. In this example we use a `request_method` attribute, which can be `GET`, `POST`, `PUT`, etc.

Let's first define a `Request` class with a `request_method` attribute:

```
class Request(object):
    def __init__(self, request_method, body=''):
        self.request_method = request_method
        self.body = body
```

We've also defined a `body` attribute which contains text in case the request is a `POST` request.

We use the previously defined `Document` as the model class.

Now we define a view function that dispatches on the class of the model instance, and the `request_method` attribute of the request:

```
@reg.dispatch(
  reg.match_instance('obj'),
  reg.match_key('request_method',
                lambda obj, request: request.request_method))
def view(obj, request):
    raise NotImplementedError
```

As you can see here we use `match_instance` and `match_key` instead of strings to specify how to dispatch.

If you use a string argument, this string names an argument and dispatch is based on the class of the instance you pass in. Here we use `match_instance`, which is equivalent to this: we have a `obj` predicate which uses the class of the `obj` argument for dispatch.

We also use `match_key`, which dispatches on the `request_method` attribute of the request; this attribute is a string, so dispatch is on string matching, not `isinstance` as with `match_instance`. You can use any Python immutable with `match_key`, not just strings.

We now define concrete views for `Document` and `Image`:

```
@view.register(request_method='GET', obj=Document)
def document_get(obj, request):
    return "Document text is: " + obj.text

@view.register(request_method='POST', obj=Document)
def document_post(obj, request):
    obj.text = request.body
    return "We changed the document"
```

Let's also define them for `Image`:

```
@view.register(request_method='GET', obj=Image)
def image_get(obj, request):
    return obj.bytes

@view.register(request_method='POST', obj=Image)
def image_post(obj, request):
    obj.bytes = request.body
    return "We changed the image"
```

Let's try it out:

```
>>> view(doc, Request('GET'))
'Document text is: Hello world!'
>>> view(doc, Request('POST', 'New content'))
'We changed the document'
>>> doc.text
'New content'
>>> view(image, Request('GET'))
'abc'
>>> view(image, Request('POST', "new data"))
'We changed the image'
>>> image.bytes
'new data'
```

1.5 Dispatch methods

Rather than having a `size` function and a `view` function, we can also have a context class with `size` and `view` as methods. We need to use *reg.dispatch_method* instead of *reg.dispatch* to do this.

```
class CMS(object):

    @reg.dispatch_method('item')
    def size(self, item):
        raise NotImplementedError

    @reg.dispatch_method(
        reg.match_instance('obj'),
        reg.match_key('request_method',
                      lambda self, obj, request: request.request_method))
    def view(self, obj, request):
        return "Generic content of {} bytes.".format(self.size(obj))
```

We can now register an implementation of `CMS.size` for a `Document` object:

```
@CMS.size.register(item=Document)
def document_size_as_method(self, item):
    return len(item.text)
```

Note that this is almost the same as the function `document_size` we defined before: the only difference is the signature, with the additional `self` as the first argument. We can in fact use *reg.methodify()* to reuse such functions without an initial context argument:

```
from reg import methodify

CMS.size.register(methodify(folder_size), item=Folder)
CMS.size.register(methodify(image_size), item=Image)
CMS.size.register(methodify(file_size), item=File)
```

`CMS.size` now behaves as expected:

```
>>> cms = CMS()
>>> cms.size(Image("123"))
3
>>> cms.size(Document("12345"))
5
```

Similarly for the `view` method we can define:

```
@CMS.view.register(request_method='GET', obj=Document)
def document_get(self, obj, request):
    return "{}-byte-long text is: {}".format(
        self.size(obj), obj.text)
```

This works as expected as well:

```
>>> cms.view(Document("12345"), Request("GET"))
'5-byte-long text is: 12345'
>>> cms.view(Image("123"), Request("GET"))
'Generic content of 3 bytes.'
```

For more about how you can use dispatch methods and class-based context, see Context-based dispatch.

1.6 Lower level API

1.6.1 Component lookup

You can look up the function that a function would dispatch to without calling it. You do this using the *reg.Dispatch.component()* method on the dispatch function:

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

Sometimes it's useful to have more control and go to a lower level by specifying the keys that go in directly. We can use *reg.Dispatch.component_by_keys()* for that:

```
>>> size.component_by_keys(item=Document) is document_size
True
```

1.6.2 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 *reg.Dispatch.all()* method 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

size.register(htmldocument_size, item=HtmlDocument)
```

`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 ...>]
```

1.6.3 Predicate key

In some cases it can be useful to get an immutable key that represents a dispatch registration. The Morepath web framework uses this for instance to determine whether registrations are identical in its conflict detection and override system.

Earlier we registered various views for object and request method. We can get immutable keys for such registrations using *reg.Dispatch.key_dict_to_predicate_key()*:

```
>>> view.key_dict_to_predicate_key(
...   {'request_method': 'GET', 'obj': Document})
(<class 'Document'>, 'GET')
>>> view.key_dict_to_predicate_key(
...   {'obj': Image, 'request_method': 'POST'})
(<class 'Image'>, 'POST')
```

Context-based dispatch

2.1 Introduction

Consider this advanced use case for Reg: we have a runtime with multiple contexts. For each context, you want the dispatch behavior to be different. Concretely, if you have an application where you can call a view dispatch function, you want it to execute a different function and return a different value in each separate context.

The Morepath web framework uses this feature of Reg to allow the developer to compose a larger application from multiple smaller ones.

You can define application context as a class. This context class defines dispatch *methods*. When you subclass the context class, you establish a new context: each subclass has entirely different dispatch registrations, and shares nothing with its base class.

2.2 A Context Class

Here is a concrete example. First we define a context class we call `A`, and a `view` dispatch method on it:

```
import reg

class A(object):
    @reg.dispatch_method(
      reg.match_instance('obj'),
      reg.match_key('request_method',
                    lambda self, obj, request: request.request_method))
    def view(self, obj, request):
        return "default"
```

Note that since `view` is a method we define a `self` argument.

To have something to view, We define `Document` and `Image` content classes:

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

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

We also need a request class:

```
class Request(object):
    def __init__(self, request_method, body=''):
        self.request_method = request_method
        self.body = body
```

To try this out, we need to create an instance of the context class:

```
a = A()
```

Before we register anything, we get the default result we defined in the method:

```
>>> doc = Document('Hello world!')
>>> a.view(doc, Request('GET'))
'default'
>>> a.view(doc, Request('POST', 'new content'))
'default'
>>> image = Image('abc')
>>> a.view(image, Request('GET'))
'default'
```

Here are the functions we are going to register:

```
def document_get(obj, request):
    return "Document text is: " + obj.text

def document_post(obj, request):
    obj.text = request.body
    return "We changed the document"

def image_get(obj, request):
    return obj.bytes

def image_post(obj, request):
    obj.bytes = request.body
    return "We changed the image"
```

We now want to register them with our context. To do so, we need to access the dispatch function through its class (`A`), not its instance (`a`). All instances of `A` (but not instances of its subclasses as we will see later) share the same registrations.

We use *`reg.methodify()`* to do the registration, to keep our view functions the same as when context is not in use. We will see an example without *`reg.methodify()`* later:

```
from reg import methodify
A.view.register(methodify(document_get),
                request_method='GET',
                obj=Document)
A.view.register(methodify(document_post),
                request_method='POST',
                obj=Document)
A.view.register(methodify(image_get),
                request_method='GET',
                obj=Image)
A.view.register(methodify(image_post),
                request_method='POST',
                obj=Image)
```

Now that we've registered some functions, we get the expected behavior when we call `a.view`:

```
>>> a.view(doc, Request('GET'))
'Document text is: Hello world!'
>>> a.view(doc, Request('POST', 'New content'))
'We changed the document'
>>> doc.text
'New content'
>>> a.view(image, Request('GET'))
'abc'
>>> a.view(image, Request('POST', "new data"))
'We changed the image'
>>> image.bytes
'new data'
```

2.3 A new context

Okay, we associate a dispatch method with a context class, but what is the point? The point is that we can introduce a new context that has different behavior now. To do, we subclass `A`:

```
class B(A):
    pass
```

At this point the new `B` context is empty of specific behavior, even though it subclasses `A`:

```
>>> b = B()
>>> b.view(doc, Request('GET'))
'default'
>>> b.view(doc, Request('POST', 'New content'))
'default'
>>> b.view(image, Request('GET'))
'default'
>>> b.view(image, Request('POST', "new data"))
'default'
```

We can now do our registrations. Let's register the same behavior for documents as we did for `Context`:

```
B.view.register(methodify(document_get),
                request_method='GET',
                obj=Document)
B.view.register(methodify(document_post),
                request_method='POST',
                obj=Document)
```

But we install *different* behavior for `Image`:

```
def b_image_get(obj, request):
    return 'New image GET'

def b_image_post(obj, request):
    return 'New image POST'

B.view.register(methodify(b_image_get),
                request_method='GET',
                obj=Image)
B.view.register(methodify(b_image_post),
                request_method='POST',
                obj=Image)
```

Calling `view` for `Document` works as before:

```
>>> b.view(doc, Request('GET'))
'Document text is: New content'
```

But the behavior for `Image` instances is different in the `B` context:

```
>>> b.view(image, Request('GET'))
'New image GET'
>>> b.view(image, Request('POST', "new data"))
'New image POST'
```

Note that the original context `A` is of course unaffected and still has the behavior we registered for it:

```
>>> a.view(image, Request('GET'))
'new data'
```

The idea is that you can create a framework around your base context class. Where this base context class needs to have dispatch behavior, you define dispatch methods. You then create different subclasses of the base context class and register different behaviors for them. This is what Morepath does with its `App` class.

2.4 Call method in the same context

What if in a dispatch implementation you find you need to call another dispatch method? How to access the context? You can do this by registering a function that get a context as its first argument. As an example, we modify our document functions so that `document_post` uses the other:

```
def c_document_get(context, obj, request):
    return "Document text is: " + obj.text

def c_document_post(context, obj, request):
    obj.text = request.body
    return "Changed: " + context.view(obj, Request('GET'))
```

Now `c_document_post` uses the `view` dispatch method on the context. We need to register these methods using *`reg.Dispatch.register()`* without *`reg.methodify()`*. This way they get the context as the first argument. Let's create a new context and do so:

```
class C(A):
    pass

C.view.register(c_document_get,
                request_method='GET',
                obj=Document)
C.view.register(c_document_post,
                request_method='POST',
                obj=Document)
```

We now get the expected behavior:

```
>>> c = C()
>>> c.view(doc, Request('GET'))
'Document text is: New content'
>>> c.view(doc, Request('POST', 'Very new content'))
'Changed: Document text is: Very new content'
```

You could have used *`reg.methodify()`* for this too, as `methodify` inspects the first argument and if it's identical to the second argument to `methodify`, it will pass in the context as that argument.

```
class D(A):
    pass

D.view.register(methodify(c_document_get, 'context'),
                request_method='GET',
                obj=Document)
D.view.register(methodify(c_document_post, 'context'),
                request_method='POST',
                obj=Document)
```

```
>>> d = D()
>>> d.view(doc, Request('GET'))
'Document text is: Very new content'
>>> d.view(doc, Request('POST', 'Even newer content'))
'Changed: Document text is: Even newer content'
```

The default value for the second argument to `methodify` is `app`.

Patterns

Here we look at a number of patterns you can implement with Reg.

3.1 Adapters

What if we wanted to add a feature that required multiple methods, not just one? You can use the adapter pattern for this.

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.

We define `Document` and `Image` content classes:

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

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

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:

```
>>> doc = Document("Hello world")
>>> 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 generically, so let's create a generic function that gives us an implementation of `Icon` back for any object:

```
import reg

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

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

```
icon.register(DocumentIcon, obj=Document)
```

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')

icon.register(ImageIcon, obj=Image)
```

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'
>>> image = Image('abc')
>>> icon(image).small()
'image_small.png'
>>> icon(image).large()
'image_large.png'
```

3.2 Service Discovery

Some applications need 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.dispatch()
def emailer():
    raise NotImplementedError
```

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

```
sent = []

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

def actual_emailer():
    return send_email

emailer.register(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.

3.3 Replacing class methods

Reg generic functions can be used to replace methods, so that you can follow the open/closed principle and add functionality to a class without modifying it. This works for instance methods, but what about `classmethod`? This takes the *class* as the first argument, not an instance. You can configure `@reg.dispatch` decorator with a special `Predicate` instance that lets you dispatch on a class argument instead of an instance argument.

Here's what it looks like:

```
@reg.dispatch(reg.match_class('cls'))
def something(cls):
    raise NotImplementedError()
```

Note the call to `match_class()` here. This lets us specify that we want to dispatch on the class, in this case we simply want the `cls` argument.

Let's use it:

```
def something_for_object(cls):
    return "Something for %s" % cls

something.register(something_for_object, cls=object)

class DemoClass(object):
    pass
```

When we now call `something()` with `DemoClass` as the first argument we get the expected output:

```
>>> something(DemoClass)
"Something for <class 'DemoClass'>"
```

This also knows about inheritance. So, you can write more specific implementations for particular classes:

```
class ParticularClass(object):
    pass

def something_particular(cls):
    return "Particular for %s" % cls

something.register(
    something_particular,
    cls=ParticularClass)
```

When we call `something` now with `ParticularClass` as the argument, then `something_particular` is called:

```
>>> something(ParticularClass)
"Particular for <class 'ParticularClass'>"
```

CHAPTER 4

API

4.1 Dispatch functions

reg.**dispatch**(**predicates*, ***kw*)

Decorator to make a function dispatch based on its arguments.

This takes the predicates to dispatch on as zero or more parameters.

Parameters

- **predicates** – sequence of *reg.Predicate* instances to do the dispatch on. You create predicates using *reg.match_instance()*, *reg.match_key()*, *reg.match_class()*, or with a custom predicate class. You can also pass in plain string argument, which is turned into a *reg.match_instance()* predicate.
- **get_key_lookup** – a function that gets a *PredicateRegistry* instance and returns a key lookup. A *PredicateRegistry* instance is itself a key lookup, but you can return a caching key lookup (such as *reg.DictCachingKeyLookup* or *reg.LruCachingKeyLookup*) to make it more efficient.

Returns a function that you can use as if it were a *reg.Dispatch* instance.

class reg.**Dispatch**(*predicates*, *callable*, *get_key_lookup*)

Dispatch function.

You can register implementations based on particular predicates. The dispatch function dispatches to these implementations based on its arguments.

Parameters

- **predicates** – a list of predicates.
- **callable** – the Python function object to register dispatch implementations for. The signature of an implementation needs to match that of this function. This function is used as a fallback implementation that is called if no specific implementations match.
- **get_key_lookup** – a function that gets a *PredicateRegistry* instance and returns a key lookup. A *PredicateRegistry* instance is itself a key lookup, but you can return a caching key lookup (such as *reg.DictCachingKeyLookup* or *reg.LruCachingKeyLookup*) to make it more efficient.

add_predicates(*predicates*)

Add new predicates.

Extend the predicates used by this predicates. This can be used to add predicates that are configured during startup time.

Note that this clears up any registered implementations.

Parameters **`predicates`** – a list of predicates to add.

`all`(**args*, ***kw*)
Lookup all functions dispatched to with args and kw.

Looks up functions for all permutations based on predicate_key, where predicate_key is constructed from args and kw.

Args varargs. Used to extract dispatch information to construct predicate_key.

Kw keyword arguments. Used to extract dispatch information to construct predicate_key.

Returns an iterable of functions.

`all_by_keys`(***kw*)
Look up all functions dispatched to using keyword arguments.

Looks up the function to dispatch to using a `key_dict`, mapping predicate name to predicate value. Returns the fallback value (default: `None`) if nothing could be found.

Kw a dictionary. key is predicate name, value is predicate value. If omitted, predicate default is used.

Returns iterable of functions being dispatched to.

`clean`()
Clean up implementations and added predicates.

This restores the dispatch function to its original state, removing registered implementations and predicates added using *`reg.Dispatch.add_predicates()`*.

`component`(**args*, ***kw*)
Lookup function dispatched to with args and kw.

Looks up the function to dispatch to using args and kw. Returns the fallback value (default: `None`) if nothing could be found.

Args varargs. Used to extract dispatch information to construct `predicate_key`.

Kw keyword arguments. Used to extract dispatch information to construct `predicate_key`.

Returns the function being dispatched to, or None.

`component_by_keys`(***kw*)
Look up function based on key_dict.

Looks up the function to dispatch to using a key_dict, mapping predicate name to predicate value. Returns the fallback value (default: `None`) if nothing could be found.

Kw key is predicate name, value is predicate value under which it was registered. If omitted, predicate default is used.

Returns the function being dispatched to, or fallback.

`fallback`(**args*, ***kw*)
Lookup fallback for args and kw.

Args varargs. Used to extract dispatch information to construct `predicate_key`.

Kw keyword arguments. Used to extract dispatch information to construct `predicate_key`.

Returns the function being dispatched to, or fallback.

key_dict_to_predicate_key(*key_dict*)
Turn a key dict into a predicate key.

Given a key dict under which an implementation function is registered, return an immutable predicate key.

Parameters **key_dict** – dict with registration information

Returns an immutable predicate key

predicate_key(**args*, ***kw*)
Construct predicate_key for function arguments.

For function arguments, construct the appropriate `predicate_key`. This is used by the dispatch mechanism to dispatch to the right function.

If the `predicate_key` cannot be constructed from `args` and `kw`, this raises a `TypeError`.

Parameters

- **args** – the varargs given to the callable.
- **kw** – the keyword arguments given to the callable.

Returns an immutable `predicate_key` based on the predicates the callable was configured with.

register(*func=None*, ***key_dict*)
Register an implementation.

If `func` is not specified, this method can be used as a decorator and the decorated function will be used as the actual `func` argument.

Parameters

- **func** – a function that implements behavior for this dispatch function. It needs to have the same signature as the original dispatch function. If this is a *`reg.DispatchMethod`*, then this means it needs to take a first context argument.
- **key_dict** – keyword arguments describing the registration, with as keys predicate name and as values predicate values.

Returns `func`.

reg.**match_key**(*name*, *func*, *fallback=None*, *default=None*)
Predicate that returns a value used for dispatching.

Name predicate name.

Func a callable that accepts the same arguments as the generic function and returns the value used for dispatching. The returned value must be of an immutable type.

Fallback the fallback value. By default it is `None`.

Default optional default value.

Returns a *`Predicate`*.

reg.**match_instance**(*name*, *func=None*, *fallback=None*, *default=None*)
Predicate that returns an instance whose class is used for dispatching.

Name predicate name.

Func a callable that accepts the same arguments as the generic function and returns the instance whose class is used for dispatching. If `None`, use a callable returning the argument with the same name as the predicate.

Fallback the fallback value. By default it is `None`.

Default optional default value.

Returns a *Predicate*.

reg.**match_class**(*name*, *func=None*, *fallback=None*, *default=None*)
Predicate that returns a class used for dispatching.

Name predicate name.

Func a callable that accepts the same arguments as the generic function and returns a class used for dispatching. If `None`, use a callable returning the argument with the same name as the predicate.

Fallback the fallback value. By default it is `None`.

Default optional default value.

Returns a *Predicate*.

class reg.**DictCachingKeyLookup**(*key_lookup*)
A key lookup that caches.

Implements the read-only API of `reg.PredicateRegistry` using a cache to speed up access.

This cache is backed by a simple dictionary so could potentially grow large if the dispatch in question can be called with a large combination of arguments that result in a large range of different predicate keys. If so, you can use `reg.LruCachingKeyLookup` instead.

Param key_lookup - the `PredicateRegistry` to cache.

class reg.**LruCachingKeyLookup**(*key_lookup*, *component_cache_size*, *all_cache_size*, *fallback_cache_size*)
A key lookup that caches.

Implements the read-only API of `reg.PredicateRegistry`, using a cache to speed up access.

The cache is LRU so won't grow beyond a certain limit, preserving memory. This is only useful if you except the access pattern to your function to involve a huge range of different predicate keys.

Param key_lookup - the `PredicateRegistry` to cache.

Parameters

- **component_cache_size** – how many cache entries to store for the `component()` method. This is also used by dispatch calls.
- **all_cache_size** – how many cache entries to store for the the `all()` method.
- **fallback_cache_size** – how many cache entries to store for the `fallback()` method.

all(*predicate_key*)
Lookup iterable of values registered for predicate_key.

Looks up values registered for all permutations of predicate_key, the most specific first.

Parameters predicate_key – an immutable predicate key, constructed for the predicates given for this key.

Returns An iterable of registered values.

component(*predicate_key*)
Lookup value in registry based on predicate_key.

If value for predicate_key cannot be found, looks up first permutation of predicate_key for which there is a value. Permutations are made according to the predicates registered for the key.

Parameters **predicate_key** – an immutable predicate key, constructed for predicates given for this key.

Returns a registered value, or `None`.

fallback(*predicate_key*)
Lookup fallback based on predicate_key.

This finds the fallback for the most specific predicate that fails to match.

Parameters **predicate_key** – an immutable predicate key, constructed for predicates given for this key.

Returns the fallback value for the most specific predicate the failed to match.

4.2 Context-specific dispatch methods

reg.**dispatch_method**(**predicates*, ***kw*)
Decorator to make a method on a context class dispatch.

This takes the predicates to dispatch on as zero or more parameters.

Parameters

- **predicates** – sequence of *Predicate* instances to do the dispatch on. You create predicates using *reg.match_instance()*, *reg.match_key()*, *reg.match_class()*, or with a custom predicate class.

  You can also pass in plain string argument, which is turned into a *reg.match_instance()* predicate.
- **get_key_lookup** – a function that gets a *PredicateRegistry* instance and returns a key lookup. A *PredicateRegistry* instance is itself a key lookup, but you can return a caching key lookup (such as *reg.DictCachingKeyLookup* or *reg.LruCachingKeyLookup*) to make it more efficient.
- **first_invocation_hook** – a callable that accepts an instance of the class in which this decorator is used. It is invoked the first time the method is invoked.

class reg.**DispatchMethod**(*predicates*, *callable*, *get_key_lookup*)

add_predicates(*predicates*)
Add new predicates.

Extend the predicates used by this predicates. This can be used to add predicates that are configured during startup time.

Note that this clears up any registered implementations.

Parameters **predicates** – a list of predicates to add.

all(**args*, ***kw*)
Lookup all functions dispatched to with args and kw.

Looks up functions for all permutations based on predicate_key, where predicate_key is constructed from args and kw.

Args varargs. Used to extract dispatch information to construct predicate_key.

Kw keyword arguments. Used to extract dispatch information to construct predicate_key.

Returns an iterable of functions.

all_by_keys(***kw*)

Look up all functions dispatched to using keyword arguments.

Looks up the function to dispatch to using a `key_dict`, mapping predicate name to predicate value. Returns the fallback value (default: `None`) if nothing could be found.

Kw a dictionary. key is predicate name, value is predicate value. If omitted, predicate default is used.

Returns iterable of functions being dispatched to.

clean()

Clean up implementations and added predicates.

This restores the dispatch function to its original state, removing registered implementations and predicates added using *reg.Dispatch.add_predicates()*.

component(**args*, ***kw*)

Lookup function dispatched to with args and kw.

Looks up the function to dispatch to using args and kw. Returns the fallback value (default: `None`) if nothing could be found.

Args varargs. Used to extract dispatch information to construct `predicate_key`.

Kw keyword arguments. Used to extract dispatch information to construct `predicate_key`.

Returns the function being dispatched to, or None.

component_by_keys(***kw*)

Look up function based on key_dict.

Looks up the function to dispatch to using a key_dict, mapping predicate name to predicate value. Returns the fallback value (default: `None`) if nothing could be found.

Kw key is predicate name, value is predicate value under which it was registered. If omitted, predicate default is used.

Returns the function being dispatched to, or fallback.

fallback(**args*, ***kw*)

Lookup fallback for args and kw.

Args varargs. Used to extract dispatch information to construct `predicate_key`.

Kw keyword arguments. Used to extract dispatch information to construct `predicate_key`.

Returns the function being dispatched to, or fallback.

key_dict_to_predicate_key(*key_dict*)

Turn a key dict into a predicate key.

Given a key dict under which an implementation function is registered, return an immutable predicate key.

Parameters **key_dict** – dict with registration information

Returns an immutable predicate key

register(*func=None*, ***key_dict*)

Register an implementation.

If `func` is not specified, this method can be used as a decorator and the decorated function will be used as the actual `func` argument.

Parameters

- **func** – a function that implements behavior for this dispatch function. It needs to have the same signature as the original dispatch function. If this is a *reg.DispatchMethod*, then this means it needs to take a first context argument.
- **key_dict** – keyword arguments describing the registration, with as keys predicate name and as values predicate values.

Returns func.

reg.**clean_dispatch_methods**(*cls*)
For a given class clean all dispatch methods.

This resets their registry to the original state using *reg.DispatchMethod.clean()*.

Parameters **cls** – a class that has *reg.DispatchMethod* methods on it.

reg.**methodify**(*func*, *selfname=None*)
Turn a function into a method, if needed.

If selfname is not specified, wrap the function so that it takes an additional first argument, like a method.

If selfname is specified, check whether it is the same as the name of the first argument of func. If itsn't, wrap the function so that it takes an additional first argument, with the name specified by selfname.

If it is, the signature of func needn't be amended, but wrapping might still be necessary.

In all cases, inspect_methodified() lets you retrieve the wrapped function.

Parameters

- **func** – the function to turn into method.
- **selfname** – if specified, the name of the argument referencing the class instance. Typically, "self".

Returns function that can be used as a method when assigned to a class.

4.3 Errors

exception reg.**RegistrationError**
Registration error.

4.4 Argument introspection

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 returns None if given something that is not callable.

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.

4.5 Predicate Registry

class reg.**PredicateRegistry**(*predicate*)

key_dict_to_predicate_key(*d*)
Construct predicate key from key dictionary.

Uses name and default attributes of predicate to construct the predicate key. If the key cannot be constructed then a KeyError is raised.

Parameters key_dict – dictionary with predicate name keys and predicate values. For omitted keys, the predicate default is used.

Returns an immutable predicate_key based on the dictionary and the names and defaults of the predicates the callable was configured with.

class reg.**Predicate**(*name*, *index*, *get_key=None*, *fallback=None*, *default=None*)
A dispatch predicate.

Parameters

- **name** – predicate name. This is used by reg.Registry.register_function_by_name().
- **index** – a function that constructs an index given a fallback argument; typically you supply either a *KeyIndex* or *ClassIndex*.
- **get_key** – a callable that accepts a dictionary with the invocation arguments of the generic function and returns a key to be used for dispatching.
- **fallback** – optional fallback value. The fallback of the the most generic index for which no values could be found is used.
- **default** – optional predicate default. This is used by reg.Registry.register_function_by_name(), and supplies the value if it is not given explicitly.

class reg.**ClassIndex**(*fallback=None*)

permutations(*key*)
Permutations for class key.

Returns class and its base classes in mro order. If a classic class in Python 2, smuggle in object as the base class anyway to make lookups consistent.

class reg.**KeyIndex**(*fallback=None*)

fallback(*key*)
Return fallback if this index does not contain key.

If index contains permutations of key, then NOT_FOUND is returned.

permutations(*key*)
Permutations for a simple immutable key.

There is only a single permutation: the key itself.

reg.**key_predicate**(*name*, *get_key=None*, *fallback=None*, *default=None*)
Construct predicate indexed on any immutable value.

Parameters

- **name** – predicate name.
- **get_key** – a callable that accepts a dictionary with the invocation arguments of the generic function and returns a key to be used for dispatching.
- **fallback** – a fallback value. By default is `None`.
- **default** – optional default value.

Returns a *Predicate*.

`reg.`**`class_predicate`**(*name*, *get_key=None*, *fallback=None*, *default=None*)
Construct predicate indexed on class.

Parameters

- **name** – predicate name.
- **get_key** – a callable that accepts a dictionary with the invocation arguments of the generic function and returns a key to be used for dispatching.
- **fallback** – a fallback value. By default is `None`.
- **default** – optional default value.

Returns a *Predicate*.

CHAPTER 5

Developing Reg

5.1 Install Reg for development

Clone Reg from github:

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

If this doesn't work and you get an error 'Permission denied (publickey)', you need to upload your ssh public key to github.

Then go to the reg directory:

```
$ cd reg
```

Make sure you have virtualenv installed.

Create a new virtualenv for Python 3 inside the reg directory:

```
$ virtualenv -p python3 env/py3
```

Activate the virtualenv:

```
$ source env/py3/bin/activate
```

Make sure you have recent setuptools and pip installed:

```
$ pip install -U setuptools pip
```

Install the various dependencies and development tools from develop_requirements.txt:

```
$ pip install -Ur develop_requirements.txt
```

For upgrading the requirements just run the command again.

If you want to test Reg with Python 2.7 as well you can create a second virtualenv for it:

```
$ virtualenv -p python2.7 env/py27
```

You can then activate it:

```
$ source env/py27/bin/activate
```

Then uprade setuptools and pip and install the develop requirements as described above.

Note: The following commands work only if you have the virtualenv activated.

5.2 Running the tests

You can run the tests using py.test:

```
$ py.test
```

To generate test coverage information as HTML do:

```
$ 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.

5.3 Running the documentation tests

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

```
(py3) $ sphinx-build -b doctest doc doc/build/doctest
```

Or alternatively if you have `Make` installed:

```
(py3) $ cd doc
(py3) $ make doctest
```

Or from the Reg project directory:

```
(py3) $ make -C doc doctest
```

Since the sample code in the documentation is maintained in Python 3 syntax, we do not support running the doctests with Python 2.7.

5.4 Building the HTML documentation

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

```
$ sphinx-build doc doc/build/html
```

Or alternatively if you have `Make` installed:

```
$ cd doc
$ make html
```

Or from the Reg project directory:

```
$ make -C doc html
```

5.5 Various checking tools

flake8 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:

```
$ flake8 reg
```

To also show cyclomatic complexity, use this command:

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

5.6 Tox

With tox you can test Morepath under different Python environments.

We have Travis continuous integration installed on Morepath's github repository and it runs the same tox tests after each checkin.

First you should install all Python versions which you want to test. The versions which are not installed will be skipped. You should at least install Python 3.5 which is required by flake8, coverage and doctests and Python 2.7 for testing Morepath with Python 2.

One tool you can use to install multiple versions of Python is pyenv.

To find out which test environments are defined for Morepath in tox.ini run:

```
$ tox -l
```

You can run all tox tests with:

```
$ tox
```

You can also specify a test environment to run e.g.:

```
$ tox -e py35
$ tox -e pep8
$ tox -e docs
```

To run a simple performance test you can use:

```
$ tox -e perf
```

CHAPTER 6

History of Reg

Reg was written by Martijn Faassen. The core mapping code was originally co-authored by Thomas Lotze, though this has since been subsumed into the generalized predicate architecture. After a few years of use, Stefano Taschini initiated a large refactoring and API redesign.

Reg is a predicate dispatch implementation for Python, with support for multiple dispatch registries in the same runtime. It was originally heavily inspired by the Zope Component Architecture (ZCA) consisting of the `zope.interface` and `zope.component` packages. Reg has strongly evolved since its inception into a general function dispatch library. Reg's codebase is completely separate from the ZCA and it has an entirely different API. At the end I've included a brief history of the ZCA.

The primary use case for Reg has been the Morepath web framework, which uses it heavily.

6.1 Reg History

The Reg code went through a quite bit of history as our insights evolved.

6.1.1 iface

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

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

This registry was instrumental in getting Reg started, but was subsequently removed in a later refactoring.

6.1.2 crom

In early 2012, Martijn 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, Martijn 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. Martijn worked a bit on Crom after the sprint, but soon moved on to other matters.

6.1.3 iface + crom

At the Plone conference held in Arnhem, the Netherlands in October 2012, Martijn gave a lightning talk about Crom, which was received positively, which reignited his interest. In the end of 2012 Martijn mailed Thomas Lotze to ask to 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.

6.1.4 ZCA-style Reg

In July 2013 in development work for CONTACT (contact.de), Martijn found himself in need of clever registries. Crom also had some configuration code intermingled with the component architecture code, and Martijn wanted to separate this out.

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

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

This version of Reg was still very similar in concepts to the Zope Component Architecture, though it used a streamlined API. This streamlined API lead to further developments.

6.1.5 Generic dispatch

After Martijn's first announcement of Reg to the world in September 2013 he got a question why it shouldn't just use PEP 443, which has a generic function implementation (single dispatch). This lead to the idea of converting Reg into a generic function implementation (with multiple dispatch), as it was already very close. After talking to some people about this at PyCon DE in october, Martijn did the refactoring to use generic functions throughout and no interfaces for lookup. Martijn then used this version of Reg in Morepath for about a year.

6.1.6 Predicate dispatch

In October 2014 Martijn had some experience with using Reg and found some of its limitations:

- Reg would try to dispatch on *all* non-keyword arguments of a function. This is not what is desired in many cases. We need a way to dispatch only on specified arguments and leave others alone.
- Reg had an undocumented predicate subsystem used to implement view lookup in Morepath. A new requirement lead to the requirement to dispatch on the class of an instance, and while Reg's generic dispatch system could do it, the predicate subsystem could not. Enabling this required a major reorganization of Reg.
- Martijn realized that such a reorganized predicate system could actually be used to generalize the way Reg worked based on how predicates worked.
- This would allow predicates to play along in Reg's caching infrastructure, which could then speed up Morepath's view lookups.
- A specific use case to replace class methods caused me to introduce `reg.classgeneric`. This could be subsumed in a generalized predicate infrastructure as well.

So in October 2014, Martijn refactored Reg once again in the light of this, generalizing the generic dispatch further to predicate dispatch, and replacing the iface-based registry. This refactoring resulted in a smaller, more unified codebase that has more features and was also faster.

6.1.7 Removing implicitness and inverting layers

Reg used an implicit `lookup` system to find the current registry to use for dispatch. This allows Morepath to compose larger applications out of smaller registries, each with their own dispatch context. As an alternative to the implicit system, you could also pass in a custom `lookup` argument to the function to indicate the current registry.

In 2016 Stefano Taschini started pushing on Morepath's use of dispatch functions and their implicit nature. Subsequent discussions with Martijn led to the insight that if we approached dispatch functions as dispatch *methods* on a context class (the Morepath application), we could get rid of the implicit behavior altogether, while gaining performance as we'd use Python's method mechanism.

In continuing discussions, Stefano also suggested that there was no need for Reg in cases where the dispatch behavior of Reg was not needed. This led to the insight that this non-dispatch behavior could be installed as methods directly on the context class.

Stefano also proposed that Reg could be internally simplified if we made the multiple registry behavior less central to the implementation, and let each dispatch function maintain its own registry. Stefano and Martijn then worked on an implementation where the dispatch method behavior is layered on top of a simpler dispatch function layer.

6.2 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, though Reg has undergone a significant evolution since then. `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.

CHANGES

7.1 0.10 (2016-10-04)

- **Breaking change**

  Reg has undergone another API breaking change. The goals of this change were:

  – Make everything explicit.

  – A simpler implementation structure – dispatch functions maintain their own registries, which allows for less interacting objects.

  – Make the advanced context-dependent dispatch more Pythonic by using classes with special dispatch methods.

  Detailed changes:

  – `reg.Registry` is gone. Instead you register directly on the dispatch function:

```
@reg.dispatch('a')
def foo(a):
  ...

def foo_implementation(a):
  ...

foo.register(foo_implementation, a=Document)
```

  – Caching is now per dispatch function, not globally per lookup. You can pass a `get_key_lookup` function that wraps `reg.PredicateRegistry` instance inside a `reg.DictCachingKeyLookup` cache. You can also use a `reg.LruCachingKeyLookup` if you expect a dispatch to be called with a large amount of possible predicate combinations, to preserve memory.

  – The whole concept of a "lookup" is gone:

    * `reg.implicit` is gone: everything is explicit. There is no more implicit lookup.

    * `reg.Lookup` itself is gone – its now implemented directly in the dispatch object, but was already how you accessed it.

    * The special `lookup` argument to pass through the current `Lookup` is gone. If you need context-dependent dispatch, you use dispatch methods.

    * If you need context dependent dispatch, where the functions being dispatched to depend on application context (such as Morepath's application mounting), you use `reg.dispatch_method` to create a dispatch method. A dispatch method maintains an entirely separate dispatch registry for each

subclass. You use `reg.methodify` to register a dispatch function that takes an optional context first argument.

If you do not use the context-dependent dispatch feature, then to upgrade your code:

- remove any `reg.set_implicit` from your code, setup of `Lookup` and the like.
- If you use an explicit `lookup` argument you can just remove them.
- You also need to change your registration code: no more `reg.Registry` setup.
- Change your registrations to be on the dispatch objects itself using `Dispatch.register`.
- To enable caching you need to set up `get_key_lookup` on the dispatch functions. You can create a partially applied version of `dispatch` to make this less verbose:

```
import reg
from functools import partial

def get_caching_key_lookup(r):
    return reg.CachingKeyLookup(r, 5000, 5000, 5000)

dispatch = partial(reg.dispatch, get_key_lookup=get_caching_key_lookup)
```

- `dispatch_external_predicates` is gone. Just use `dispatch` directly. You can add predicates to an existing Dispatch object using the `add_predicates` method.

If you do use the context-dependent dispatch feature, then you also need to:

- identify the context class in your application (or create one).
- move the dispatch functions to this class, marking them with `@reg.dispatch_method` instead of `@reg.dispatch`.
- Registration is now using <context_class>.<method>.register. Functions you register this way behave as methods to `context_class`, so get an instance of this class as the first argument.
- You can also use `reg.methodify` to register implementation functions that do not take the context as the first argument – this is useful when upgrading existing code.
- Call your context-dependent methods as methods on the context instance. This way you can indicate what context you are calling your dispatch methods in, instead of using the *lookup*' argument.

In some cases you want a context-dependent method that actually does not dispatch on any of its arguments. To support this use case you can simply set function (that takes an app argument) as a the method on the context class directly:

```
Context.my_method = some_function
```

If you want to set up a function that doesn't take a reference to a `Context` instance as its first argument, you can use `reg.methodify` to turn it into a method that ignores its first argument:

```
Context.my_method = reg.methodify(some_function)
```

If you want to register a function that might or might not have a reference to a `Context` instance as its first argument, called, e.g., `app`, you can use the following:

```
Context.my_method = reg.methodify(some_function, selfname='app')
```

- **Breaking change**

  Removed the helper function `mapply` from the API.

- **Breaking change**

  Removed the exception class `KeyExtractorError` from the API. When passing the wrong number of arguments to a dispatch function, or when using the wrong argument names, you will now get a TypeError, in conformity with standard Python behaviour.

- **Breaking change**

  Removed the `KeyExtractor` class from the API. Callables used in predicate construction now expect the same arguments as the dispatch function.

- **Breaking change**

  Removed the `argnames` attribute from `Predicate` and its descendant.

- **Breaking change**

  Remove the `match_argname` predicate. You can now use `match_instance` with no callable instead.

- The second argument for `match_class` is now optional; if you don't supply it `match_class` will generate a predicate function that extracts that name by default.
- The second argument for `match_instance` is now optional; if you don't supply it `match_instance` will generate a predicate function that extracts that name by default.
- Include doctests in Tox and Travis.
- We now use virtualenv and pip instead of buildout to set up the development environment. The development documentation has been updated accordingly.
- As we reached 100% code coverage for pytest, coveralls integration was replaced by the `--fail-under=100` argument of `coverage report` in the tox coverage test.

7.2 0.9.3 (2016-07-18)

- Minor fixes to documentation.
- Add tox test environments for Python 3.4 and 3.5, PyPy 3 and PEP 8.
- Make Python 3.5 the default Python environment.
- Changed location `NoImplicitLookupError` was imported from in `__init__.py`.

7.3 0.9.2 (2014-11-13)

- Reg was a bit too strict; when you had multiple (but not single) predicates, Reg would raise KeyError when you put in an unknown key. Now they're just being silently ignored, as they don't do any harm.
- Eliminated a check in `ArgExtractor` that could never take place.
- Bring test coverage back up to 100%.
- Add converage configuration to ignore test files in coverage reporting.

7.4 0.9.1 (2014-11-11)

- A bugfix in the behavior of the fallback logic. In situations with multiple predicates of which one is a class predicate it was possible for a fallback not to be found even though a fallback was available.

7.5 0.9 (2014-11-11)

Total rewrite of Reg! This includes a range of changes that can break code. The primary motivations for this rewrite:

- unify predicate system with class-based lookup system.
- extract dispatch information from specific arguments instead of all arguments.

Some specific changes:

- Replaced `@reg.generic` decorator with `@reg.dispatch()` decorator. This decorator can be configured with predicates that extract information from the arguments. Rewrite this:

```
@reg.generic
def foo(obj):
   pass
```

  to this:

```
@reg.dispatch('obj')
def foo(obj):
   pass
```

  And this:

```
@reg.generic
def bar(a, b):
    pass
```

  To this:

```
@reg.dispatch('a', 'b')
def bar(a, b):
    pass
```

  This is to get dispatch on the classes of these instance arguments. If you want to match on the class of an attribute of an argument (for instance) you can use `match_instance` with a function:

```
@reg.dispatch(match_instance('a', lambda a: a.attr))
```

  The first argument to `match_instance` is the name of the predicate by which you refer to it in `register_function`.

  You can also use `match_class` to have direct dispatch on classes (useful for replicating classmethods), and `match_key` to have dispatch on the (immutable) value of the argument (useful for a view predicate system). Like for `match_instance`, you supply functions to these match functions that extract the exact information to dispatch on from the argument.

- The `register_function` API replaces the `register` API to register a function. Replace this:

```
r.register(foo, (SomeClass,), dispatched_to)
```

  with:

```
r.register_function(foo, dispatched_to, obj=SomeClass)
```

  You now use keyword parameters to indicate exactly those arguments specified by `reg.dispatch()` are actually predicate arguments. You don't need to worry about the order of predicates anymore when you register a function for it.

- The new `classgeneric` functionality is part of the predicate system now; you can use `reg.match_class` instead. Replace:

```
@reg.classgeneric
def foo(cls):
   pass
```

with:

```
@reg.dispatch(reg.match_class('cls', lambda cls: cls))
def foo(cls):
    pass
```

You can do this with any argument now, not just the first one.

- pep443 support is gone. Reg is focused on its own dispatch system.
- Compose functionality is gone – it turns out Morepath doesn't use lookup composition to support App inheritance. The cached lookup functionality has moved into `registry.py` and now also supports caching of predicate-based lookups.
- Dependency on the future module is gone in favor of a small amount of compatibility code.

7.6 0.8 (2014-08-28)

- Added a `@reg.classgeneric`. This is like `@reg.generic`, but the first argument is treated as a class, not as an instance. This makes it possible to replace `@classmethod` with a generic function too.
- Fix documentation on running documentation tests. For some reason this did not work properly anymore without running sphinxpython explicitly.
- Optimization: improve performance of generic function calls by employing `lookup_mapply` instead of general `mapply`, as we only care about passing in the lookup argument when it's defined, and any other arguments should work as before. Also added a `perf.py` which is a simple generic function timing script.

7.7 0.7 (2014-06-17)

- Python 2.6 compatibility. (Ivo van der Wijk)
- Class maps (and thus generic function lookup) now works with old style classes as well.
- Marked as production/stable now in `setup.py`.

7.8 0.6 (2014-04-08)

- Removed unused code from mapply.py.
- Typo fix in API docs.

7.9 0.5 (2014-01-21)

- Make `reg.ANY` public. Used for predicates that match any value.

7.10 0.4 (2014-01-14)

- arginfo has been totally rewritten and is now part of the public API of reg.

7.11 0.3 (2014-01-06)

- Experimental Python 3.3 support thanks to the future module.

7.12 0.2 (2013-12-19)

- If a generic function implementation defines a `lookup` argument that argument will be the lookup used to call it.
- Added `reg.mapply()`. This allows you to call things with more keyword arguments than it accepts, ignoring those extra keyword args.
- A function that returns `None` is not assumed to fail, so no fallback to the original generic function is triggered anymore.
- An optional `precalc` facility is made available on `Matcher` to avoid some recalculation.
- Implement a specific `PredicateMatcher` that matches a value on predicate.

7.13 0.1 (2013-10-28)

- Initial public release.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

r