Reg Documentation

Release 0.9.3

Martijn Faassen

July 18, 2016

Contents

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 that dispatch on some of their arguments. To support this, Reg provides an implementation of multiple dispatch in Python. Reg goes beyond dispatching on the class of the arguments but can also dispatch on other aspects of arguments.

In other words: 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 implementations 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. 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. The Morepath web framework is built on top of Reg. 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 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"
```

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

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

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

In `TitledReport` there's an attribute called `title` but in the `LabeledReport` 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*, and tell it about a few implementations for the `title` function:

```
registry = reg.Registry()
registry.register_function(
    title, titled_report_title, obj=TitledReport)
registry.register_function(
    title, labeled_report_title, obj=LabeledReport)
```

We then tell Reg to use it automatically using *reg.implicit.Implicit.initialize()*:

```
from reg import implicit
implicit.initialize(registry.lookup())
```

Once we've done this, our 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'
```

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 examine a situation where generic functions are useful.

CHAPTER 2

Generic functions

2.1 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:

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

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

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:

```python
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
```

Note that 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
```

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(item):
    return len(item.text)

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

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(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 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(item):
    return len(item.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(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
```

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

```
r = reg.Registry()
r.register_function(size, document_size, item=Document)
r.register_function(size, folder_size, item=Folder)
r.register_function(size, image_size, item=Image)
r.register_function(size, file_size, item=File)
```

We can now use our `size` function:

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

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

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

```
r.register_function(icon, 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')

r.register_function(icon, 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'
>>> 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.dispatch('request', 'model')
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_function(view, document_view,
                    request=Request, model=Document)
```

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 call `view`:

```
>>> 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.dispatch()
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_function(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.

3.1 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', lambda cls: 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, and we supply a lambda function that shows how to extract this from the arguments to `something`; in this case we simply want the `cls` argument.

Let's use it:

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

r.register_function(something, 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

r.register_function(something, 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'>"
```

Lower level API

4.1 Component lookup

You can look up the function that a function would dispatch to without calling it. You do this using the `component` method on the dispatch function:

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

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_function(size, 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 ...>]
```

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 register predicate for any immutable key such as a string:

```
r.register_predicates('some key', [reg.match_argname('obj')])
```

We can now register something for this key:

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

We can't get it at it using a generic dispatch function anymore now. We can use the *reg.Registry* API instead. Here's what to do:

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

4.4 Caching

We can turn a plain *reg.Registry* into a faster, caching class lookup using *reg.CachingKeyLookup*:

```
>>> caching = reg.CachingKeyLookup(r, 100, 100, 100)
```

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

```
>>> caching_lookup = caching.lookup()
>>> size(doc, lookup=caching_lookup)
12
>>> size(doc, lookup=caching_lookup)
12
```

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

CHAPTER 5

API

`reg.`**`dispatch`**`(`**predicates*`)`

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 *`Predicate`* instances to do the dispatch on.

`reg.`**`match_key`**`(`*name*, *func*, *fallback=None*, *default=None*`)`

Predicate that extracts immutable key according to func.

Name predicate name.

Func argument that takes arguments. These arguments are extracted from the arguments given to the dispatch function. This function should return what to dispatch on.

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

Default optional default value.

Returns a *`Predicate`*.

`reg.`**`match_instance`**`(`*name*, *func*, *fallback=None*, *default=None*`)`

Predicate that extracts class of instance returned by func.

Name predicate name.

Func argument that takes arguments. These arguments are extracted from the arguments given to the dispatch function. This function should return an instance; dispatching is done on the class of that instance.

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

Default optional default value.

Returns a *`Predicate`*.

`reg.`**`match_argname`**`(`*argname*, *fallback=None*, *default=None*`)`

Predicate that extracts class of specified argument.

Name predicate name.

Argname name of the argument to dispatch on - its class will be used for the dispatch.

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

Default optional default value.

Returns a *`Predicate`*.

reg.**match_class**(*name*, *func*, *fallback=None*, *default=None*)

Predicate that extracts class returned by func.

Name predicate name.

Func argument that takes arguments. These arguments are extracted from the arguments given to the dispatch function. This function should return a class; dispatching is done on this class.

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

Default optional default value.

Returns a *Predicate*.

class reg.**Lookup**(*key_lookup*)

The lookup is used for generic dispatch.

The arguments to the lookup functions are those of the dispatch function being called. The lookup extract the predicate_key from these arguments and then looks up the actual function to call. This function is then called with the original arguments.

Parameters **key_lookup** – the key lookup, either a *Registry* or *CachingKeyLookup*.

all(*callable*, **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.

Parameters **callable** – the dispatch function.

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_key_dict(*callable*, *key_dict*)

Look up all functions dispatched to using 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.

Parameters **callable** – the dispatch function

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

Returns iterable of functions being dispatched to.

call(*callable*, **args*, ***kw*)

Call callable with args and kw.

Do a dispatch call for callable. If nothing more specific is registered, call the dispatch function as a fallback.

Parameters **callable** – the dispatch function to call.

Args varargs for the call. Is also used to extract dispatch information to construct predicate_key.

Kw keyword arguments for the call. Is also used to extract dispatch information to construct predicate_key.

Returns the result of the call.

`component` (*callable*, **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.

Parameters **`callable`** – the dispatch function.

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_key_dict` (*callable*, *key_dict*)

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.

Parameters **`callable`** – the dispatch function

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

Returns the function being dispatched to, or fallback.

`fallback` (*callable*, **args*, ***kw*)

Lookup fallback for args and kw.

Parameters **`callable`** – the dispatch function.

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.

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

class `reg.`**`Predicate`** (*name*, *index*, *get_key=None*, *fallback=None*, *default=None*)

A dispatch predicate.

`argnames` ()

argnames that this predicate needs to dispatch on.

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.

Name predicate name.

Get_key a `KeyExtractor`. Should return key to dispatch on.

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.

Name predicate name.

Get_key a `KeyExtractor`. Should return class to dispatch on.

Fallback a fallback value. By default is `None`.

Default optional default value.

Returns a *`Predicate`*.

class `reg.`**`Registry`**

A registry of predicate registries

The key is an immutable that we perform the lookup for. The permutation key is the key to do the lookup with. The value should be an immutable as well (or at least hashable).

The registry can be configured with predicates for a key, and the predicates are aware of permutations of keys. This means, among others, that a value registered for a base class is also matched when you look up a subclass.

The Registry is designed to be easily cacheable.

`all`(*key*, *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

- **`key`** – an immutable for which to look up the values.
- **`predicate_key`** – an immutable predicate key, constructed for the predicates given for this key.

Returns An iterable of registered values.

`clear`()

Clear the registry.

`component`(*key*, *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

- **`key`** – an immutable for which to look up the predicate_key.
- **`predicate_key`** – an immutable predicate key, constructed for predicates given for this key.

Returns a registered value, or `None` if nothing was found.

fallback(*key*, *predicate_key*)

Lookup fallback based on predicate_key.

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

Parameters

- **key** – an immutable for which to look up the predicate_key.
- **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.

key_dict_to_predicate_key(*callable*, *key_dict*)

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

- **callable** – the callable for which to extract the predicate_key
- **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.

lookup()

A *Lookup* for this registry.

predicate_key(*callable*, **args*, ***kw*)

Construct predicate_key for function arguments.

For a callable and its 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 *KeyExtractorError*.

Parameters

- **callable** – the callable for which to extract the predicate_key
- **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_callable_predicates(*callable*, *predicates*)

Register predicates for a callable key.

Works as *register_predicates()*, but also makes sure a predicate key can be constructed from arguments to this callable.

Parameters

- **callable** – a function from which to extract argument information.
- **predicates** – a sequence of *reg.Predicate* objects.

Returns a *reg.PredicateRegistry*.

`register_dispatch`(*callable*)
Register a dispatch function.

Works as *`register_dispatch_predicates()`*, but extracts predicate information from information registered.

For *`dispatch()`* these are the predicates supplied to it in its arguments.

For `dispatch_external_predicates()` these are the predicates supplied using *`register_external_predicates()`*.

Parameters **`callable`** – a dispatch callable.

Returns a *`reg.PredicateRegistry`*.

`register_dispatch_predicates`(*callable*, *predicates*)
Register a dispatch function.

Works as *`register_callable_predicates()`*, but works with a dispatch and makes sure predicates can't be registered twice.

Parameters

- **`callable`** – a dispatch callable.
- **`predicates`** – a sequence of *`reg.Predicate`* objects.

Returns a *`reg.PredicateRegistry`*.

`register_external_predicates`(*callable*, *predicates*)
Register external predicates for dispatch_external_predicates.

dispatch_external_predicates looks for predicates registered for the dispatch here. You can define them here.

Parameters

- **`callable`** – a dispatch_external_predicates callable.
- **`predicates`** – a sequence of *`reg.Predicate`* objects.

`register_function`(*callable*, *value*, ***key_dict*)
Register a callable for a dispatch function.

Like *`register_function_by_predicate_key()`*, but constructs the predicate_key based on the `key_dict` argument, using the `name` and `default` arguments to *`Predicate`*.

`register_function_by_predicate_key`(*callable*, *predicate_key*, *value*)
Register a callable for a dispatch function.

Like *`register_value()`*, but makes sure that the value is a callable with the same signature as the original dispatch callable. If not, a *`reg.RegistrationError`* is raised.

`register_predicates`(*key*, *predicates*)
Register the predicates to use for a lookup key.

Parameters

- **`key`** – an immutable for which to register the predicates.
- **`predicates`** – a sequence of *`reg.Predicate`* objects.

Returns a *`reg.PredicateRegistry`*.

`register_value`(*key*, *predicate_key*, *value*)
Register a value for a predicate_key.

Given a key, register a value for a particular predicate_key in the registry. Raises a `reg.RegistrationError` if the predicate_key is already known for this key.

Parameters

- **`key`** – an immutable
- **`predicate_key`** – an immutable predicate key defined by the predicates for this key.
- **`value`** – an immutable value to register.

class `reg.`**`CachingKeyLookup`**(*key_lookup*, *component_cache_size*, *all_cache_size*, *fallback_cache_size*)

A key lookup that caches.

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

The cache is LRU.

Param key_lookup - the `Registry` 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`(*key*, *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

- **`key`** – an immutable for which to look up the values.
- **`predicate_key`** – an immutable predicate key, constructed for the predicates given for this key.

Returns An iterable of registered values.

`component`(*key*, *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

- **`key`** – an immutable for which to look up the predicate_key.
- **`predicate_key`** – an immutable predicate key, constructed for predicates given for this key.

Returns a registered value, or `None`.

`fallback`(*key*, *predicate_key*)

Lookup fallback based on predicate_key.

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

Parameters

- **`key`** – an immutable for which to look up the predicate_key.

- **`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.

`lookup`()
A *`Lookup`* for this registry.

class `reg.`**`Lookup`**(*key_lookup*)
The lookup is used for generic dispatch.

The arguments to the lookup functions are those of the dispatch function being called. The lookup extract the predicate_key from these arguments and then looks up the actual function to call. This function is then called with the original arguments.

Parameters **`key_lookup`** – the key lookup, either a *`Registry`* or *`CachingKeyLookup`*.

`all`(*callable*, **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.

Parameters **`callable`** – the dispatch function.

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_key_dict`(*callable*, *key_dict*)
Look up all functions dispatched to using 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.

Parameters **`callable`** – the dispatch function

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

Returns iterable of functions being dispatched to.

`call`(*callable*, **args*, ***kw*)
Call callable with args and kw.

Do a dispatch call for callable. If nothing more specific is registered, call the dispatch function as a fallback.

Parameters **`callable`** – the dispatch function to call.

Args varargs for the call. Is also used to extract dispatch information to construct predicate_key.

Kw keyword arguments for the call. Is also used to extract dispatch information to construct predicate_key.

Returns the result of the call.

`component`(*callable*, **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.

Parameters **`callable`** – the dispatch function.

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_key_dict`(*callable*, *key_dict*)
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.

Parameters **`callable`** – the dispatch function

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

Returns the function being dispatched to, or fallback.

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

Parameters **`callable`** – the dispatch function.

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.

exception `reg.`**`RegistrationError`**
Registration error.

exception `reg.`**`KeyExtractorError`**
A lookup key could not be constructed.

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`.

Dispatch 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.

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 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.

CHAPTER 6

Developing Reg

6.1 Install Reg for development

First make sure you have virtualenv installed for Python 3.5.

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.

6.2 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.

6.3 Running the documentation tests

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

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

6.4 Building the HTML documentation

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

```
$ bin/sphinxbuilder
```

6.5 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 7

History of Reg

Reg was written by me, Martijn Faassen. The core mapping code was originally co-authored by Thomas Lotze, though this has since been subsumed into the generalized predicate architecture.

Reg is a generic 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.

7.1 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. I used this version of Reg in Morepath for about a year.

In October 2014 I 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. I realized that it could not be properly used to dispatch on the class of an instance without reorganizing Reg in a major way.
- I 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, I refactored Reg once again in the light of this. This results in a smaller, more unified codebase that has more features and is hopefully also faster.

7.2 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.

CHANGES

8.1 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`.

8.2 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.

8.3 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.

8.4 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.

8.5 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.

8.6 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`.

8.7 0.6 (2014-04-08)

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

8.8 0.5 (2014-01-21)

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

8.9 0.4 (2014-01-14)

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

8.10 0.3 (2014-01-06)

- Experimental Python 3.3 support thanks to the future module.

8.11 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.

8.12 0.1 (2013-10-28)

- Initial public release.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Python Module Index