beanbag Documentation

Release 1.9.2

Anthony Towns

March 31, 2015

Contents

BeanBag is a set of modules that provide some syntactic sugar to make interacting with REST APIs easy and pleasant.

A simple example:

```
>>> import beanbag  # version 1 api
>>> github = beanbag.BeanBag("https://api.github.com")
>>> watchers = github.repos.ajtowns.beanbag.watchers()
>>> for w in watchers:
...     print(w["login"])

>>> import beanbag.v2 as beanbag # version 2 api
>>> github = beanbag.BeanBag("https://api.github.com")
>>> watchers = GET(github.repos.ajtowns.beanbag.watchers)
>>> for w in watchers:
...     print(w.login)
```

CHAPTER 1

Contents

1.1 beanbag.v2 – REST API access

A quick example:

```
>>> from beanbag.v2 import BeanBag, GET
>>> gh = BeanBag("https://api.github.com/")
>>> watchers = GET(gh.repos.ajtowns.beanbag.watchers)
>>> for w in watchers:
...     print(w.login)
```

Setup:

```
>>> import beanbag.v2 as beanbag
>>> from beanbag.v2 import GET, POST, PUT, PATCH, DELETE
>>> myapi = beanbag.BeanBag("http://hostname/api/")
```

To constuct URLs, you can use attribute-style access or dict-style access:

```
>>> print(myapi.foo)
http://hostname/api/foo
>>> print(myapi["bar"])
http://hostname/api/bar
```

You can chain paths as well:

```
>>> print(myapi.foo.bar["baz"][3].xyzzy)
http://hostname/api/foo/bar/baz/3/xyzzy
```

To do a request on a resource that requires a trailing slash:

```
>>> print(myapi.foo._)
http://hostname/api/foo/
>>> print(myapi.foo[""])
http://hostname/api/foo/
>>> print(myapi.foo["/"])
http://hostname/api/foo/
>>> print(myapi["foo/"])
http://hostname/api/foo/
>>> print(myapi.foo._.x == myapi.foo.x)
True
>>> print(myapi.foo["_"])
http://hostname/api/foo/_
```

You can add URL parameters using function calls:

```
>>> print(myapi.foo(a=1, b="foo"))
http://hostname/api/foo?a=1;b=foo
```

Finally, to actually do REST queries on these queries you can use the GET, POST, PUT, PATCH and DELETE functions. The first argument should be a BeanBag url, and the second argument (if provided) should be the request body, which will be json encoded before being sent. The return value is the request's response (decoded from json).

```
>>> res = GET( foo.resource )
>>> res = POST( foo.resource, {"a": 12} )
>>> DELETE( foo.resource )
```

To access REST interfaces that require authentication, you need to specify a session object when instantiating the BeanBag initially. BeanBag supplies helpers to make Kerberos and OAuth 1.0a authentication easier.

1.1.1 BeanBag class

The BeanBag class does all the magic described above, using `beanbag.namespace`.

class `beanbag.v2.`**BeanBag**(*base_url*, *ext=''*, *session=None*, *use_attrdict=True*)

__init__(*base_url*, *ext=''*, *session=None*, *use_attrdict=True*)
Create a BeanBag referencing a base REST path.

Parameters

- **base_url** – the base URL prefix for all resources
- **ext** – extension to add to resource URLs, eg ".json"
- **session** – requests.Session instance used for this API. Useful to set an auth procedure, or change verify parameter.
- **use_attrdict** – if true, `decode()` will wrap dicts and lists in a `beanbag.attrdict.AttrDict` for syntactic sugar.

__call__(**args*, ***kwargs*)
Set URL parameters

__eq__(*other*)
self == other

__getattr__(*attr*)
self.attr

__getitem__(*item*)
self[attr]

__invert__()
Provide access to the base/path via the namespace object

```
bb = BeanBag(...)
base, path = ~bb.foo
assert isinstance(base, BeanBagBase)
```

This is the little bit of glue needed so that it's possible to call methods defined in BeanBagBase directly rather than just the operators BeanBag supports.

__ne__(*other*)
self != other

__repr__()
Human readable representation of object

__str__()
Obtain the URL of a resource

This class can be subclassed. In particular, if you need to use something other than JSON for requests or responses, subclassing `BeanBagBase` and overriding the `encode` and `decode` methods is probably what you want to do. One caveat: due to the way `beanbag.namespace` works, if you wish to invoke the parent classes method, you'll usually need the parent *base* class, accessed via `~BeanBag` or `super(~SubClass, self)`.

1.1.2 HTTP Verbs

Functions are provided for the standard set of HTTP verbs.

beanbag.v2.**GET**(*url*, *body=None*)
GET verb function

beanbag.v2.**HEAD**(*url*, *body=None*)
HEAD verb function

beanbag.v2.**POST**(*url*, *body=None*)
POST verb function

beanbag.v2.**PUT**(*url*, *body=None*)
PUT verb function

beanbag.v2.**PATCH**(*url*, *body=None*)
PATCH verb function

beanbag.v2.**DELETE**(*url*, *body=None*)
DELETE verb function

The verb function is used to create BeanBag compatible verbs. It is used as:

```
GET = verb("GET")
```

beanbag.v2.**verb**(*verbname*)
Construct a BeanBag compatible verb function

Parameters **verbname** – verb to use (GET, POST, etc)

1.1.3 Request

The `Request` class serves as a place holder for arguments to `requests.Session.request`. Normally this is constructed from a dict object passed to `POST` or `PUT` via `json.dumps()` however a `Request` object can also be created by hand and passed in as the `body` parameter in a `POST` or similar `BeanBag` request. For example to make a POST request with a body that isn't valid JSON:

```
POST(bbexample.path.to.resource, Request(body="MAGIC STRING"))
```

This can be useful with GET requests as well, even though GET requests don't have a body per se:

```
GET(bbexample.path.to.resource, Request(headers={"X-Magic": "String"}))
```

class beanbag.v2.**Request**(***kwargs*)
Bases: beanbag.attrdict.AttrDict

__init__(***kwargs*)
Create a Request object

Request objects act as placeholders for the arguments to the requests() function of the requests.Session being used. They are used as the interface between the encode() and make_request() functions, and may also be used by the API caller.

NB: A Request object is only suitable for one use, as it may be modified in-place during the request. For this reason, __init__ makes a (shallow) copy of all the keyword arguments supplied rather than using them directly.

1.1.4 BeanBagException

exception beanbag.v2.**BeanBagException**(*response*, *msg*)
Exception thrown when a BeanBag request fails.

Data members:

- msg – exception string, brief and human readable
- response – response object

You can get the original request via bbe.response.request.

__init__(*response*, *msg*)
Create a BeanBagException

1.2 beanbag.v1 – Original-style REST API access

Setup:

```
>>> import beanbag.v1 as beanbag
>>> foo = beanbag.BeanBag("http://hostname/api/")
```

To do REST queries, then:

```
>>> r = foo.resource(p1=3.14, p2=2.718)   # GET request
>>> r = foo.resource( {"a": 3, "b": 7} ) # POST request
>>> del foo.resource                      # DELETE request
>>> foo.resource = {"a" : 7, "b": 3}      # PUT request
>>> foo.resource += {"a" : 7, "b": 3}     # PATCH request
```

You can chain paths as well:

```
>>> print(foo.bar.baz[3]["xyzzy"].q)
http://hostname/api/foo/bar/baz/3/xyzzy/q
```

To do a request on a resource that requires a trailing slash:

```
>>> print(foo.bar._)
http://hostname/api/foo/bar/
>>> print(foo.bar[""])
http://hostname/api/foo/bar/
>>> print(foo.bar["/"])
http://hostname/api/foo/bar/
>>> print(foo["bar/"])
http://hostname/api/foo/bar/
>>> print(foo.bar._.x == foo.bar.x)
```

```
True
>>> print(foo.bar["_"])
http://hostname/api/foo/bar/_
```

To access REST interfaces that require authentication, you need to specify a session object. BeanBag supplies helpers to make Kerberos and OAuth 1.0a authentication easier.

To setup oauth using OAuth1 directly:

```
>>> import requests
>>> from requests_oauth import OAuth1
>>> session = requests.Session()
>>> session.auth = OAuth1( consumer creds, user creds )
>>> foo = beanbag.BeanBag("http://hostname/api/", session=session)
```

Using the `OAuth10aDance` helper is probably a good plan though.

1.2.1 BeanBag class

class `beanbag.v1.`**BeanBag**(*base_url*, *ext=''*, *session=None*, *fmt='json'*)

__init__(*base_url*, *ext=''*, *session=None*, *fmt='json'*)
Create a BeanBag referencing a base REST path.

Parameters

- **base_url** – the base URL prefix for all resources
- **ext** – extension to add to resource URLs, eg ".json"
- **session** – requests.Session instance used for this API. Useful to set an auth procedure, or change verify parameter.
- **fmt** – either 'json' for json data, or a tuple specifying a content-type string, encode function (for encoding the request body) and a decode function (for decoding responses)

__call__(**args*, ***kwargs*)
Make a GET, POST or generic request to a resource.

Example

```
>>> x = BeanBag("http://host/api")
>>> r = x()                                  # GET request
>>> r = x(p1='foo', p2=3)                    # GET request with parameters passed via query s
>>> r = x( {'a': 1, 'b': 2} )                # POST request
>>> r = x( "RANDOMIZE", {'a': 1, 'b': 2} )   # Custom HTTP verb with request body
>>> r = x( "OPTIONS", None )                 # Custom HTTP verb with empty request body
```

__delattr__(*attr*)
del self.attr

__delitem__(*item*)
del self[item]

__getattr__(*attr*)
self.attr

__getitem__(*item*)
self[attr]

__iadd__(*val*)
Make a PATCH request to a resource.

Example

```
>>> x = BeanBag("http://host/api")
>>> x += {"op": "replace", "path": "/a", "value": 3}
```

__setattr__(*attr*, *val*)
self.attr = val

__setitem__(*item*, *val*)
self[item] = val

__str__()
Obtain the URL of a resource

1.2.2 BeanBagException

exception beanbag.v1.**BeanBagException**(*response*, *msg*)
Exception thrown when a BeanBag request fails.

Data members:

- msg – exception string, brief and human readable
- response – response object

You can get the original request via bbe.response.request.

__init__(*response*, *msg*)
Create a BeanBagException

1.3 beanbag.auth – Authentication Helpers

1.3.1 Kerberos Helper

To setup kerberos auth:

```
>>> import requests
>>> session = requests.Session()
>>> session.auth = beanbag.KerbAuth()
>>> foo = beanbag.BeanBag("http://hostname/api/", session=session)
```

class beanbag.auth.**KerbAuth**(*timeout=180*)
Helper class for basic Kerberos authentication using requests library. A single instance can be used for multiple sites. Each request to the same site will use the same authorization token for a period of 180 seconds.

Example

```
>>> session = requests.Session()
>>> session.auth = KerbAuth()
```

__init__(*timeout=180*)

1.3.2 OAuth 1.0a Helper

`OAuth10aDance` helps with determining the user creds, compared to using OAuth1 directly.

class `beanbag.auth.`**`OAuth10aDance`**(*req_token=None*, *acc_token=None*, *authorize=None*, *client_key=None*, *client_secret=None*, *user_key=None*, *user_secret=None*)

`__init__`(*req_token=None*, *acc_token=None*, *authorize=None*, *client_key=None*, *client_secret=None*, *user_key=None*, *user_secret=None*)

Create an OAuth10aDance object to negotiatie OAuth 1.0a credentials.

The first set of parameters are the URLs to the OAuth 1.0a service you wish to authenticate against.

Parameters

- **req_token** – Request token URL
- **authorize** – User authorization URL
- **acc_token** – Access token URL

These parameters (and the others) may also be provided by subclassing the OAuth10aDance class, eg:

Example

```
>>> class OAuthDanceTwitter(beanbag.OAuth10aDance):
...     req_token = "https://api.twitter.com/oauth/request_token"
...     authorize = "https://api.twitter.com/oauth/authorize"
...     acc_token = "https://api.twitter.com/oauth/access_token"
```

The second set of parameters identify the client application to the server, and need to be obtained outside of the OAuth protocol.

Parameters

- **client_key** – client/consumer key
- **client_secret** – client/consumer secret

The final set of parameters identify the user to server. These may be left as None, and obtained using the OAuth 1.0a protocol via the `obtain_creds()` method or using the `get_auth_url()` and `verify_user()` methods.

Parameters

- **user_key** – user key
- **user_secret** – user secret

Assuming OAuthDanceTwitter is defined as above, and you have obtained the client key and secret (see https://apps.twitter.com/ for twitter) as `k` and `s`, then putting these together looks like:

Example

```
>>> oauthdance = OAuthDanceTwitter(client_key=k, client_secret=s)
>>> oauthdance.obtain_creds()
Please go to url:
  https://api.twitter.com/oauth/authorize?oauth_token=...
  Please input the verifier: 1111111
>>> session = requests.Session()
>>> session.auth = oauthdance.oauth()
```

`have_creds`()

Check whether all credentials are filled in

get_auth_url()
URL for user to obtain verification code

verify_user(*verifier*)
Set user key and secret based on verification code

obtain_creds()
Fill in credentials by interacting with the user (input/print)

oauth()
Create an OAuth1 authenticator using client and user credentials

1.4 beanbag.attrdict – Access dict members by attribute

1.4.1 AttrDict

This module provides the `AttrDict` class, which allows you to access dict members via attribute access, allowing similar syntax to javascript objects. For example:

```
d = {"foo": 1, "bar": {"sub": {"subsub": 2}}}
ad = AttrDict(d)
assert ad["foo"] == ad["foo"]
assert ad.foo == 1
assert ad.bar.sub.subsub == 2
```

Note that `AttrDict` simply provides a view on the native dict. That dict can be obtained using the plus operator like so:

```
ad = AttrDict(d)
assert +ad is d
```

This allows use of native dict methods such as `d.update()` or `d.items()`. Note that attribute access binds more tightly than plus, so brackets will usually need to be used, eg: `(+ad.bar).items()`.

An `AttrDict` can also be directly used as an iterator (`for key in attrdict: ...`) and as a container (`if key in attrdict: ...`).

class beanbag.attrdict.**AttrDict**(*base=None*)

__delattr__(*attr*)
del self.attr

__delitem__(*item*)
del self[item]

__eq__(*other*)
self == other

__getattr__(*attr*)
self.attr

__getitem__(*item*)
self[attr]

__init__(*base=None*)
Provide an AttrDict view of a dictionary.

Parameters **base** – dictionary/list to be viewed

__ne__(*other*)
self != other

__pos__()
View underlying dict object

__setattr__(*attr*, *val*)
self.attr = val

__setitem__(*item*, *val*)
self[item] = val

1.5 beanbag.namespace

The beanbag.namespace module allows defining classes that provide arbitrary namespace behaviour. This is what allows the other beanbag modules to provide their clever syntactic sugar.

A entry in a namespace is identified by two components: a base and a path. The base is constructed once for a namespace and is common to all entries in the namespace, and each entry's path is used to differentiate them. For example, with `AttrDict`, the base is the underlying dictionary (`d`), while the path is the sequence of references into that dictionary (eg, `("foo", "bar")` corresponding to `d["foo"]["bar"]`). The reason for splitting these apart is mostly efficiency – the path element needs to be cheap and easy to construct and copy since that may need to happen for an attribute access.

To define a namespace you provide a class that inherits from `beanbag.namespace.Namespace` and defines the methods the base class should have. The `NamespaceMeta` metaclass then creates a new base class containing these methods, and builds the namespace class on top of that base class, mapping Python's special method names to the corresponding base class methods, minus the underscores. For example, to define the behavour of the ~ operator (aka `__invert__(self)`), the Base class defines a method:

```
def invert(self, path):
    ...
```

The code can rely on the base value being `self`, and the path being `path`, then do whatever calculation is necessary to create a result. If that result should be a different entry in the same namespace, that can be created by invoking `self.namespace(newpath)`.

In order to make inplace operations work more smoothly, returning `None` from those options will be automatically treated as returning the original namespace object (ie `self.namespace(path)`, without the overhead of reconstructing the object). This is primarily to make it easier to avoid the "double setting" behaviour of python's inplace operations, ie where `a[i] += j` is converted into:

```
tmp = a.__getitem__(i)    # tmp = a[i]
res = tmp.__iadd__(j)     # tmp += j
a.__setitem__(i, res)     # a[i] = tmp
```

In particular, implementations of `setitem` and `setattr` can avoid poor behaviour here by testing whether the value being set (`res`) is already the existing value, and performing a no-op if so. The `SettableHierarchialNS` class implements this behaviour.

1.5.1 NamespaceMeta

The `NamespaceMeta` metaclass provides the magic for creating arbitrary namespaces from Base classes as discussed above. When set as the metaclass for a class, it will turn a base class into a namespace class directly, while constructing an appropriate base class for the namespace to use.

class beanbag.namespace.**NamespaceMeta**

__invert__()
Obtain base class for namespace

__module__ = 'beanbag.namespace'

static __new__(*mcls*, *name*, *bases*, *nmspc*)

classmethod deferfn(*mcls*, *cls*, *nsdict*, *basefnname*, *inum=False*, *attr=False*)

classmethod make_namespace(*mcls*, *cls*)
create a unique Namespace class based on provided class

ops = ['repr', 'str', 'call', 'bool', 'getitem', 'setitem', 'delitem', 'len', 'iter', 'reversed', 'contains', 'enter', 'exit', 'pos', 'ne

ops_attr = ['getattr', 'setattr', 'delattr']

ops_inum = ['iadd', 'isub', 'imul', 'ipow', 'idiv', 'ifloordiv', 'ilshift', 'irshift', 'iand', 'ior', 'ixor']

static wrap_path_fn(*basefn*)

static wrap_path_fn_attr(*basefn*)

static wrap_path_fn_inum(*basefn*)

1.5.2 NamespaceBase

The generated base class will inherit from `NamespaceBase` (or the base class corresponding to any namespaces the namespace class inherits from), and will have a `Namespace` attribute referencing the namespace class. Further, the generated base class can be accessed by using the inverse opertor on the namespace class, ie `MyNamespaceBase = ~MyNamespace`.

class beanbag.namespace.**NamespaceBase**
Base class for user-defined namespace classes' bases

Namespace = None
Replaced in subclasses by the corresponding namespace class

namespace(*path=None*)
Used to create a new Namespace object from the Base class

1.5.3 Namespace

`Namespace` provides a trivial Base implementation. It's primarily useful as a parent class for inheritance, so that you don't have explicitly set `NamespaceMeta` as your metaclass.

class beanbag.namespace.**Namespace**(**args*, ***kwargs*)

1.5.4 HierarchialNS

`HierarchialNS` provides a simple basis for producing namespaces with freeform attribute and item hierarchies, eg, where you might have something like `ns.foo.bar["baz"]`.

By default, this class specifies a path as a tuple of attributes, but this can be changed by overriding the `path` and `_get` methods. If some conversion is desired on either attribute or item access, the `attr` and `item` methods can be overridden respectively.

Otherwise, to get useful behaviour from this class, you probably want to provide some additional methods, such as `__call__`.

class `beanbag.namespace.`**`HierarchialNS`**
Bases: `beanbag.namespace.Namespace`

`__eq__`(*other*)
self == other

`__getattr__`(*attr*)
self.attr

`__getitem__`(*item*)
self[attr]

`__init__`()

`__module__` = **'beanbag.namespace'**

`__ne__`(*other*)
self != other

`__repr__`()
Human readable representation of object

`__str__`()
Returns path joined by dots

1.5.5 SettableHierarchialNS

`SettableHierarchialNS` is intended to make life slightly easier if you want to be able to assign to your hierarchial namespace. It provides `set` and `delete` methods that you can implement, without having to go to the trouble of implementing both item and attribute variants of both functions.

This class implements the check for "setting to self" mentioned earlier in order to prevent inplace operations having two effects. It uses the `eq` method to test for equality.

class `beanbag.namespace.`**`SettableHierarchialNS`**(**args*, ***kwargs*)
Bases: `beanbag.namespace.HierarchialNS`

`__delattr__`(*attr*)
del self.attr

`__delitem__`(*item*)
del self[item]

`__eq__`(*other*)
self == other

`__getattr__`(*attr*)
self.attr

`__getitem__`(*item*)
self[attr]

`__init__`(**args*, ***kwargs*)

`__module__` = **'beanbag.namespace'**

`__ne__`(*other*)
self != other

__repr__()
Human readable representation of object

__setattr__(*attr*, *val*)
self.attr = val

__setitem__(*item*, *val*)
self[item] = val

__str__()
Returns path joined by dots

1.5.6 sig_adapt

This is a helper function to make that generated methods in the namespace object provide more useful help.

beanbag.namespace.**sig_adapt**(*sigfn*, *dropargs=None*, *name=None*)
Function decorator that changes the name and (optionally) signature of a function to match another function. This is useful for making the help of generic wrapper functions match the functions they're wrapping. For example:

```
def foo(a, b, c, d=None):
    pass

@sig_adapt(foo)
def myfn(*args, **kwargs):
    pass
```

The optional "name" parameter allows renaming the function to something different to the original function's name.

The optional "dropargs" parameter allows dropping arguments by position or name. (Note positions are 0 based, so to convert foo(self, a, b) to foo(a, b) specify dropargs=("self",) or dropargs=(0,))

1.6 Examples

What follows are some examples of using BeanBag for various services.

1.6.1 GitHub

GitHub's REST API, using JSON for data and either HTTP Basic Auth or OAuth2 for authentication. Basic Auth is perfect for a command line app, since the user can just use their github account password directly.

The following example uses the github API to list everyone who's starred one of your repos, and which repo it is that they've starred.

```
#!/usr/bin/env python

import beanbag.v1 as beanbag
import os
import requests

sess = requests.Session()
sess.auth = (os.environ["GITHUB_ACCT"], os.environ["GITHUB_PASS"])
```

```
github = beanbag.BeanBag("https://api.github.com/", session=sess)

myuser = github.user()
me = myuser["login"]
repos = github.users[me].repos()

repo = {}
who = {}

for r in repos:
    rn = r["name"]
    repo[rn] = github.repos[me][rn]()
    stars = github.repos[me][rn].stargazers()
    for s in stars:
        sn = s["login"]
        if sn not in who:
            who[sn] = set()
        who[sn].add(rn)

for w in sorted(who):
    print("%s:" % (w,))
    for rn in sorted(who[w]):
        print("  %s -- %s" % (rn, repo[rn]["description"]))
```

1.6.2 Twitter

Twitter's REST API is slightly more complicated. It still uses JSON, but requires OAuth 1.0a to be used for authentication. OAuth is designed primarily for webapps, where the application is controlled by a third party. In particular it is designed to allow an "application" to authenticate as "authorised by a particular user", rather than allowing the application to directly authenticate itself as the user (eg, by using the user's username and password directly, as we did above with github).

This in turn means that the application has to be able to identify itself. This is done by gaining "client credential", in Twitter's case via Twitter Apps.

The process of having an application to ask a user to provide a token that allows it to access Twitter on behalf of the user is encapsulated in the `OAuth10aDance` class. In the example below is subclassed in order to provide the Twitter-specific URLs that the user and application will need to visit in order to gain the right tokens to do the authentication. The `obtain_creds()` method is called, which will instruct the user to enter any necessary credentials, after which a `Session` object is created and setup to perform OAuth authentication using the provided credentials.

The final minor complication is that Twitter's endpoints all end with ".json", which would be annoying to have to specify via beanbag (since "." is not a valid part of an attribute). The `ext=` keyword argument of the `BeanBag` constructor is used to supply this as the standard extension for all URLs in the Twitter API.

```
#!/usr/bin/env python

import beanbag
import requests

class OAuthDanceTwitter(beanbag.OAuth10aDance):
    req_token = "https://api.twitter.com/oauth/request_token"
    authorize = "https://api.twitter.com/oauth/authorize"
    acc_token = "https://api.twitter.com/oauth/access_token"

client_key, client_secret = (None, None)
user_key, user_secret = (None, None)
```

```
oauthDance = OAuthDanceTwitter(
        client_key=client_key, client_secret=client_secret,
        user_key=user_key, user_secret=user_secret)
oauthDance.obtain_creds()

session = requests.Session()
session.auth = oauthDance.oauth()

twitter = beanbag.BeanBag("https://api.twitter.com/1.1/", ext=".json",
                          session=session)

myacct = twitter.account.settings()
me = myacct["screen_name"]
tweets = twitter.statuses.user_timeline(screen_name=me, count=7)
for tweet in tweets:
    print(repr(tweet["text"]))
```

CHAPTER 2

Credits

Code contributors:

- Anthony Towns <aj@erisian.com.au>
- Gary Martin <gjm@apache.org>
- Lubos Kocman <lkocman@redhat.com>
- Daniel Mach <dmach@redhat.com>

Documentation contributors:

- Anthony Towns <aj@erisian.com.au>

Test case contributors and bug reporters:

- Anthony Towns <aj@erisian.com.au>
- Russell Stuart <russell-github@stuart.id.au>

BeanBag is inspired by Kadir Pekel's Hammock, though sadly only shares a license, and not any actual code. Hammock is available from *https://github.com/kadirpekel/hammock*.

CHAPTER 3

Indices and tables

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

b

Symbols