WebHelpers Documentation

Release 1.3

Ben Bangert

November 17, 2011

CONTENTS

All helpers are organized into subpackages.

CHAPTER

ONE

WEBHELPERS.CONSTANTS

Place names and other constants often used in web forms.

1.1 Countries

webhelpers.constants.**country_codes**()
Return a list of all country names as tuples. The tuple value is the country's 2-letter ISO code and its name; e.g., `("GB", "United Kingdom")`. The countries are in name order.

Can be used like this:

```
import webhelpers.constants as constants
from webhelpers.html.tags import select
select("country", country_codes(),
    prompt="Please choose a country ...")
```

See here for more information: http://www.iso.org/iso/english_country_names_and_code_elements

1.2 States & Provinces

webhelpers.constants.**us_states**()
List of USA states.

Return a list of `(abbreviation, name)` for all US states, sorted by name. Includes the District of Columbia.

webhelpers.constants.**us_territories**()
USA postal abbreviations for territories, protectorates, and military.

The return value is a list of `(abbreviation, name)` tuples. The locations are sorted by name.

webhelpers.constants.**canada_provinces**()
List of Canadian provinces.

Return a list of `(abbreviation, name)` tuples for all Canadian provinces and territories, sorted by name.

`webhelpers.constants.`**`uk_counties`**`()`
Return a list of UK county names.

1.3 Deprecations

The timezone helpers were removed in WebHelpers 0.6. Install the PyTZ package if you need them.

CHAPTER

TWO

WEBHELPERS.CONTAINERS

Container objects, and helpers for lists and dicts.

This would have been called this "collections" except that Python 2 can't import a top-level module that's the same name as a module in the current package.

2.1 Classes

class webhelpers.containers.**Counter**

I count the number of occurrences of each value registered with me.

Call the instance to register a value. The result is available as the `.result` attribute. Example:

```
>>> counter = Counter()
>>> counter("foo")
>>> counter("bar")
>>> counter("foo")
>>> sorted(counter.result.items())
[('bar', 1), ('foo', 2)]

>> counter.result
{'foo': 2, 'bar': 1}
```

To see the most frequently-occurring items in order:

```
>>> counter.get_popular(1)
[(2, 'foo')]
>>> counter.get_popular()
[(2, 'foo'), (1, 'bar')]
```

Or if you prefer the list in item order:

```
>>> counter.get_sorted_items()
[('bar', 1), ('foo', 2)]
```

classmethod correlate(*class_*, *iterable*)

Build a Counter from an iterable in one step.

This is the same as adding each item individually.

```
>>> counter = Counter.correlate(["A", "B", "A"])
>>> counter.result["A"]
2
>>> counter.result["B"]
1
```

get_popular(*max_items=None*)
Return the results as as a list of `(count, item)` pairs, with the most frequently occurring items first.

If `max_items` is provided, return no more than that many items.

get_sorted_items()
Return the result as a list of `(item, count)` pairs sorted by item.

class webhelpers.containers.**Accumulator**
Accumulate a dict of all values for each key.

Call the instance to register a value. The result is available as the `.result` attribute. Example:

```
>>> bowling_scores = Accumulator()
>>> bowling_scores("Fred", 0)
>>> bowling_scores("Barney", 10)
>>> bowling_scores("Fred", 1)
>>> bowling_scores("Barney", 9)
>>> sorted(bowling_scores.result.items())
[('Barney', [10, 9]), ('Fred', [0, 1])]

>> bowling_scores.result
{'Fred': [0, 1], 'Barney': [10, 9]}
```

The values are stored in the order they're registered.

Alternatives to this class include `paste.util.multidict.MultiDict` in Ian Bicking's Paste package.

classmethod correlate(*class_*, *iterable*, *key*)
Create an Accumulator based on several related values.

`key` is a function to calculate the key for each item, akin to `list.sort(key=)`.

This is the same as adding each item individually.

class webhelpers.containers.**UniqueAccumulator**
Accumulate a dict of unique values for each key.

The values are stored in an unordered set.

Call the instance to register a value. The result is available as the `.result` attribute.

class webhelpers.containers.**defaultdict**(*missing_func*)
A dict that automatically creates values for missing keys. This is the same as

`collections.defaultdict` in the Python standard library. It's provided here for Python 2.4, which doesn't have that class.

When you try to read a key that's missing, I call `missing_func` without args to create a value. The result is inserted into the dict and returned. Many Python type constructors can be used as `missing_func`. Passing `list` or `set` creates an empty dict or set. Passing `int` creates the integer `0`. These are useful in the following ways:

```
>> d = defaultdict(list);  d[ANYTHING].append(SOMEVALUE)
>> d = defaultdict(set);   d[ANYTHING].include(SOMEVALUE)
>> d = defaultdict(int);   d[ANYTHING] += 1
```

class webhelpers.containers.**DumbObject**(***kw*)

A container for arbitrary attributes.

Usage:

```
>>> do = DumbObject(a=1, b=2)
>>> do.b
2
```

Alternatives to this class include `collections.namedtuple` in Python 2.6, and `formencode.declarative.Declarative` in Ian Bicking's FormEncode package. Both alternatives offer more features, but `DumbObject` shines in its simplicity and lack of dependencies.

2.2 Functions

webhelpers.containers.**correlate_dicts**(*dicts*, *key*)

Correlate several dicts under one superdict.

If you have several dicts each with a 'name' key, this puts them in a container dict keyed by name.

```
>>> d1 = {"name": "Fred", "age": 41}
>>> d2 = {"name": "Barney", "age": 31}
>>> flintstones = correlate_dicts([d1, d2], "name")
>>> sorted(flintstones.keys())
['Barney', 'Fred']
>>> flintstones["Fred"]["age"]
41
```

If you're having trouble spelling this method correctly, remember: "relate" has one 'l'. The 'r' is doubled because it occurs after a prefix. Thus "correlate".

webhelpers.containers.**correlate_objects**(*objects*, *attr*)

Correlate several objects under one dict.

If you have several objects each with a 'name' attribute, this puts them in a dict keyed by name.

```
>>> class Flintstone(DumbObject):
...     pass
...
>>> fred = Flintstone(name="Fred", age=41)
>>> barney = Flintstone(name="Barney", age=31)
>>> flintstones = correlate_objects([fred, barney], "name")
>>> sorted(flintstones.keys())
['Barney', 'Fred']
>>> flintstones["Barney"].age
31
```

If you're having trouble spelling this method correctly, remember: "relate" has one 'l'. The 'r' is doubled because it occurs after a prefix. Thus "correlate".

webhelpers.containers.**del_quiet**(*dic*, *keys*)

Delete several keys from a dict, ignoring those that don't exist.

This modifies the dict in place.

```
>>> d ={"A": 1, "B": 2, "C": 3}
>>> del_quiet(d, ["A", "C"])
>>> d
{'B': 2}
```

webhelpers.containers.**distribute**(*lis*, *columns*, *direction*, *fill=None*)

Distribute a list into a N-column table (list of lists).

`lis` is a list of values to distribute.

`columns` is an int greater than 1, specifying the number of columns in the table.

`direction` is a string beginning with "H" (horizontal) or "V" (vertical), case insensitive. This affects how values are distributed in the table, as described below.

`fill` is a value that will be placed in any remaining cells if the data runs out before the last row or column is completed. This must be an immutable value such as `None` , `""`, 0, " ", etc. If you use a mutable value like `[]` and later change any cell containing the fill value, all other cells containing the fill value will also be changed.

The return value is a list of lists, where each sublist represents a row in the table. `table[0]` is the first row. `table[0][0]` is the first column in the first row. `table[0][1]` is the second column in the first row.

This can be displayed in an HTML table via the following Mako template:

```
<table>
% for row in table:
  <tr>
% for cell in row:
    <td>${cell}</td>
% endfor   cell
  </tr>
% endfor   row
</table>
```

In a horizontal table, each row is filled before going on to the next row. This is the same as dividing the list into chunks:

```
>>> distribute([1, 2, 3, 4, 5, 6, 7, 8], 3, "H")
[[1, 2, 3], [4, 5, 6], [7, 8, None]]
```

In a vertical table, the first element of each sublist is filled before going on to the second element. This is useful for displaying an alphabetical list in columns, or when the entire column will be placed in a single <td> with a
 between each element:

```
>>> food = ["apple", "banana", "carrot", "daikon", "egg", "fish", "gelato",
>>> table = distribute(food, 3, "V", "")
>>> table
[['apple', 'daikon', 'gelato'], ['banana', 'egg', 'honey'], ['carrot', 'fish
>>> for row in table:
...     for item in row:
...         print "%-9s" % item,
...     print "."    # To show where the line ends.
...
apple     daikon    gelato    .
banana    egg       honey     .
carrot    fish                .
```

Alternatives to this function include a NumPy matrix of objects.

webhelpers.containers.**except_keys**(*dic*, *keys*)

Return a copy of the dict without the specified keys.

```
>>> except_keys({"A": 1, "B": 2, "C": 3}, ["A", "C"])
{'B': 2}
```

webhelpers.containers.**extract_keys**(*dic*, *keys*)

Return two copies of the dict. The first has only the keys specified. The second has all the *other* keys from the original dict.

```
>> extract_keys({"From": "F", "To": "T", "Received", R"}, ["To", "From"])
({"From": "F", "To": "T"}, {"Received": "R"})
>>> regular, extra = extract_keys({"From": "F", "To": "T", "Received": "R"},
>>> sorted(regular.keys())
['From', 'To']
>>> sorted(extra.keys())
['Received']
```

webhelpers.containers.**only_some_keys**(*dic*, *keys*)

Return a copy of the dict with only the specified keys present.

`dic` may be any mapping. The return value is always a Python dict.

```
>> only_some_keys({"A": 1, "B": 2, "C": 3}, ["A", "C"])
>>> sorted(only_some_keys({"A": 1, "B": 2, "C": 3}, ["A", "C"]).items())
[('A', 1), ('C', 3)]
```

webhelpers.containers.**ordered_items**(*dic*, *key_order*, *other_keys=True*, *default=<class 'webhelpers.misc.NotGiven'>*)

Like `dict.iteritems()` but with a specified key order.

Arguments:

- `dic` is any mapping.
- `key_order` is a list of keys. Items will be yielded in this order.
- `other_keys` is a boolean.
- `default` is a value returned if the key is not in the dict.

This yields the items listed in `key_order`. If a key does not exist in the dict, yield the default value if specified, otherwise skip the missing key. Afterwards, if `other_keys` is true, yield the remaining items in an arbitrary order.

Usage:

```
>>> dic = {"To": "you", "From": "me", "Date": "2008/1/4", "Subject": "X"}
>>> dic["received"] = "..."
>>> order = ["From", "To", "Subject"]
>>> list(ordered_items(dic, order, False))
[('From', 'me'), ('To', 'you'), ('Subject', 'X')]
```

webhelpers.containers.**get_many**(*d*, *required=None*, *optional=None*, *one_of=None*)

Extract values from a dict for unpacking into simple variables.

`d` is a dict.

`required` is a list of keys that must be in the dict. The corresponding values will be the first elements in the return list. Raise KeyError if any of the keys are missing.

`optional` is a list of optional keys. The corresponding values will be appended to the return list, substituting None for missing keys.

`one_of` is a list of alternative keys. Take the first key that exists and append its value to the list. Raise KeyError if none of the keys exist. This argument will append exactly one value if specified, or will do nothing if not specified.

Example:

```
uid, action, limit, offset = get_many(request.params,
    required=['uid', 'action'], optional=['limit', 'offset'])
```

Contributed by Shazow.

webhelpers.containers.**transpose**(*array*)

Turn a list of lists sideways, making columns into rows and vice-versa.

`array` must be rectangular; i.e., all elements must be the same length. Otherwise the behavior is undefined: you may get `IndexError` or missing items.

Examples:

```
>>> transpose([["A", "B", "C"], ["D", "E", "F"]])
[['A', 'D'], ['B', 'E'], ['C', 'F']]
>>> transpose([["A", "B"], ["C", "D"], ["E", "F"]])
[['A', 'C', 'E'], ['B', 'D', 'F']]
>>> transpose([])
[]
```

Here's a pictoral view of the first example:

```
A B C    =>    A D
D E F          B E
               C F
```

This can be used to turn an HTML table into a group of div columns. An HTML table is row major: it consists of several <tr> rows, each containing several <td> cells. But a layout consists of only one row, each containing an entire subarray. The s have style "float:left", which makes them appear horizontally. The items within each are placed in their own 's or separated by
, which makes them appear vertically. The point is that an HTML table is row major (`array[0]` is the first row), while a group of div columns is column major (`array[0]` is the first column). `transpose()` can be used to switch between the two.

`webhelpers.containers.`**`unique`**`(`*`it`*`)`

Return a list of unique elements in the iterable, preserving the order.

Usage:

```
>>> unique([None, "spam", 2, "spam", "A", "spam", "spam", "eggs", "spam"])
[None, 'spam', 2, 'A', 'eggs']
```

CHAPTER

THREE

WEBHELPERS.DATE

Date and time helpers.

webhelpers.date.**distance_of_time_in_words**(*from_time*, *to_time=0*, *granularity='second'*, *round=False*)

Return the absolute time-distance string for two datetime objects, ints or any combination you can dream of.

If times are integers, they are interpreted as seconds from now.

`granularity` dictates where the string calculation is stopped. If set to seconds (default) you will receive the full string. If another accuracy is supplied you will receive an approximation. Available granularities are: 'century', 'decade', 'year', 'month', 'day', 'hour', 'minute', 'second'

Setting `round` to true will increase the result by 1 if the fractional value is greater than 50% of the granularity unit.

Examples:

```
>>> distance_of_time_in_words(86399, round=True, granularity='day')
'1 day'
>>> distance_of_time_in_words(86399, granularity='day')
'less than 1 day'
>>> distance_of_time_in_words(86399)
'23 hours, 59 minutes and 59 seconds'
>>> distance_of_time_in_words(datetime(2008,3,21, 16,34),
... datetime(2008,2,6,9,45))
'1 month, 15 days, 6 hours and 49 minutes'
>>> distance_of_time_in_words(datetime(2008,3,21, 16,34),
... datetime(2008,2,6,9,45), granularity='decade')
'less than 1 decade'
>>> distance_of_time_in_words(datetime(2008,3,21, 16,34),
... datetime(2008,2,6,9,45), granularity='second')
'1 month, 15 days, 6 hours and 49 minutes'
```

webhelpers.date.**time_ago_in_words**(*from_time*, *granularity='second'*, *round=False*)

Return approximate-time-distance string for `from_time` till now.

Same as `distance_of_time_in_words` but the endpoint is now.

CHAPTER

FOUR

WEBHELPERS.FEEDGENERATOR

This is a port of Django's feeed generator for creating RSS and Atom feeds. The Geo classes for publishing geographical (GIS) data are also ported. Syndication feed generation library – used for generating RSS, etc.

Sample usage:

```
>>> import webhelpers.feedgenerator as feedgenerator
>>> feed = feedgenerator.Rss201rev2Feed(
...     title=u"Poynter E-Media Tidbits",
...     link=u"http://www.poynter.org/column.asp?id=31",
...     description=u"A group weblog by the sharpest minds in online media/journ
...     language=u"en",
... )
>>> feed.add_item(title="Hello", link=u"http://www.holovaty.com/test/", descript
>>> fp = open('test.rss', 'w')
>>> feed.write(fp, 'utf-8')
>>> fp.close()
```

For definitions of the different versions of RSS, see: http://diveintomark.org/archives/2004/02/04/incompatible-rss

4.1 Classes

class webhelpers.feedgenerator.**SyndicationFeed**(*title*, *link*, *description*, *language=None*, *author_email=None*, *author_name=None*, *author_link=None*, *subtitle=None*, *categories=None*, *feed_url=None*, *feed_copyright=None*, *feed_guid=None*, *ttl=None*, ***kwargs*)

Base class for all syndication feeds. Subclasses should provide write()

add_item(*title*, *link*, *description*, *author_email=None*, *author_name=None*, *author_link=None*, *pubdate=None*, *comments=None*, *unique_id=None*, *enclosure=None*, *categories=()*, *item_copyright=None*, *ttl=None*, ***kwargs*)

Adds an item to the feed. All args are expected to be Python Unicode objects except pubdate, which is a datetime.datetime object, and enclosure, which is an instance of the Enclosure class.

add_item_elements(*handler*, *item*)

Add elements on each item (i.e. item/entry) element.

add_root_elements(*handler*)

Add elements in the root (i.e. feed/channel) element. Called from write().

item_attributes(*item*)

Return extra attributes to place on each item (i.e. item/entry) element.

latest_post_date()

Returns the latest item's pubdate. If none of them have a pubdate, this returns the current date/time.

num_items()

root_attributes()

Return extra attributes to place on the root (i.e. feed/channel) element. Called from write().

write(*outfile*, *encoding*)

Outputs the feed in the given encoding to outfile, which is a file-like object. Subclasses should override this.

writeString(*encoding*)

Returns the feed in the given encoding as a string.

class webhelpers.feedgenerator.**Enclosure**(*url*, *length*, *mime_type*)

Represents an RSS enclosure

class webhelpers.feedgenerator.**RssFeed**(*title*, *link*, *description*, *language=None*, *author_email=None*, *author_name=None*, *author_link=None*, *subtitle=None*, *categories=None*, *feed_url=None*, *feed_copyright=None*, *feed_guid=None*, *ttl=None*, ***kwargs*)

add_root_elements(*handler*)

endChannelElement(*handler*)

rss_attributes()

write(*outfile*, *encoding*)

write_items(*handler*)

class webhelpers.feedgenerator.**RssUserland091Feed**(*title*, *link*, *description*, *language=None*, *author_email=None*, *author_name=None*, *author_link=None*, *subtitle=None*, *categories=None*, *feed_url=None*, *feed_copyright=None*, *feed_guid=None*, *ttl=None*, ***kwargs*)

add_item_elements(*handler*, *item*)

class webhelpers.feedgenerator.**Rss201rev2Feed**(*title*, *link*, *description*, *language=None*, *author_email=None*, *author_name=None*, *author_link=None*, *subtitle=None*, *categories=None*, *feed_url=None*, *feed_copyright=None*, *feed_guid=None*, *ttl=None*, ***kwargs*)

add_item_elements(*handler*, *item*)

class webhelpers.feedgenerator.**Atom1Feed**(*title*, *link*, *description*, *language=None*, *author_email=None*, *author_name=None*, *author_link=None*, *subtitle=None*, *categories=None*, *feed_url=None*, *feed_copyright=None*, *feed_guid=None*, *ttl=None*, ***kwargs*)

add_item_elements(*handler*, *item*)

add_root_elements(*handler*)

root_attributes()

write(*outfile*, *encoding*)

write_items(*handler*)

`DefaultFeed` is an alias for `Rss201rev2Feed`.

4.2 Functions

webhelpers.feedgenerator.**rfc2822_date**(*date*)

webhelpers.feedgenerator.**rfc3339_date**(*date*)

webhelpers.feedgenerator.**get_tag_uri**(*url*, *date*)
Creates a TagURI. See http://diveintomark.org/archives/2004/05/28/howto-atom-id

4.3 GIS subclasses

These classes allow you to include geometries (e.g., latitude/longitude) in your feed. The implementation is in a mixin class:

class webhelpers.feedgenerator.**GeoFeedMixin**
This mixin provides the necessary routines for SyndicationFeed subclasses to produce simple GeoRSS or W3C Geo elements.

Subclasses recognize a `geometry` keyword argument to `.add_item()`. The value may be any of several types:

- a 2-element tuple or list of floats representing latitude/longitude: `(X, Y)`. This is called a "point".

- a 4-element tuple or list of floats representing a box: `(X0, Y0, X1, Y1)`.
- a tuple or list of two points: `( (X0, Y0), (X1, Y1) )`.
- a `Geometry` instance. (Or any compatible class.) This provides limited support for points, lines, and polygons. Read the `Geometry` docstring and the source of `GeoFeedMixin.add_georss_element()` before using this.

The mixin provides one class attribute:

is_input_latitude_first
The default value False indicates that input data is in latitude/longitude order. Change to True if the input data is longitude/latitude. The output is always written latitude/longitude to conform to the GeoRSS spec.

The reason for this attribute is that the Django original stores data in longitude/latutude order and reverses the arguments before writing. WebHelpers does not do this by default, but if you're using Django data or other data that has longitude first, you'll have to set this.

Methods:

add_georss_element(*handler*, *item*, *w3c_geo=False*)
This routine adds a GeoRSS XML element using the given item and handler.

add_georss_point(*handler*, *coords*, *w3c_geo=False*)
Adds a GeoRSS point with the given coords using the given handler. Handles the differences between simple GeoRSS and the more popular W3C Geo specification.

georss_coords(*coords*)
In GeoRSS coordinate pairs are ordered by lat/lon and separated by a single white space. Given a tuple of coordinates, this will return a unicode GeoRSS representation.

Two concrete subclasses are provided:

class `webhelpers.feedgenerator.`**GeoAtom1Feed**(*title*, *link*, *description*, *language=None*, *author_email=None*, *author_name=None*, *author_link=None*, *subtitle=None*, *categories=None*, *feed_url=None*, *feed_copyright=None*, *feed_guid=None*, *ttl=None*, ***kwargs*)

class webhelpers.feedgenerator.**W3CGeoFeed**(*title*, *link*, *description*, *language=None*, *author_email=None*, *author_name=None*, *author_link=None*, *subtitle=None*, *categories=None*, *feed_url=None*, *feed_copyright=None*, *feed_guid=None*, *ttl=None*, ***kwargs*)

A minimal geometry class is included:

class webhelpers.feedgenerator.**Geometry**(*geom_type*, *coords*)

A basic geometry class for `GeoFeedMixin`.

Instances have two public attributes:

geom_type

"point", "linestring", "linearring", "polygon"

coords

For **point**, a tuple or list of two floats: `(X, Y)`.

For **linestring** or **linearring**, a string: `"X0 Y0 X1 Y1 ..."`.

For **polygon**, a list of strings: `["X0 Y0 X1 Y1 ..."]`. Only the first element is used because the Geo classes support only the exterior ring.

The constructor does not check its argument types.

This class was created for WebHelpers based on the interface expected by `GeoFeedMixin.add_georss_element()`. The class is untested. Please send us feedback on whether it works for you.

CHAPTER

FIVE

WEBHELPERS.HTML

CHAPTER

SIX

WEBHELPERS.HTML.BUILDER

6.1 Classes

class webhelpers.html.builder.**HTML**
Described above.

6.2 Functions

webhelpers.html.builder.**url_escape**(*s*, *safe='/'*)
Urlencode the path portion of a URL. This is the same function as urllib.quote in the Python standard library. It's exported here with a name that's easier to remember.

The markupsafe package has a function soft_unicode which converts a string to Unicode if it's not already. Unlike the Python builtin unicode(), it will not convert Markup (literal) to plain Unicode, to avoid overescaping. This is not included in WebHelpers but you may find it useful.

CHAPTER

SEVEN

WEBHELPERS.HTML.CONVERTERS

CHAPTER

EIGHT

WEBHELPERS.HTML.GRID

A demo is available. Run the following command to produce some HTML tables:

```
python -m webhelpers.html.grid_demo OUTPUT_DIR
```

A subclass specialized for Pylons is in `webhelpers.pylonslib.grid`.

8.1 `Grid` class

CHAPTER

NINE

WEBHELPERS.HTML.TAGS

9.1 Form tags

9.2 ModelTags class

9.3 Hyperlinks

9.4 Table tags

9.5 Other non-form tags

webhelpers.html.tags.**BR**
A break tag (“
”) followed by a newline. This is a literal constant, not a function.

9.6 Head tags and document type

9.7 Utility functions

CHAPTER

TEN

WEBHELPERS.HTML.TOOLS

CHAPTER

ELEVEN

WEBHELPERS.MEDIA

Multimedia helpers for images, etc.

webhelpers.media.**choose_height**(*new_width*, *width*, *height*)
Return the height corresponding to `new_width` that's proportional to the original size (`width` x `height`).

webhelpers.media.**get_dimensions_pil**(*path*, *default=(None*, *None)*)
Get an image's size using the Python Imaging Library (PIL).

`path` is the path of the image file.

`default` is returned if the size could not be ascertained. This usually means the file does not exist or is not in a format recognized by PIL.

The normal return value is a tuple: `(width, height)`.

Depends on the Python Imaging Library. If your application is not otherwise using PIL, see the `get_dimensions()` function, which does not have external dependencies.

webhelpers.media.**get_dimensions**(*path*, *default=(None*, *None)*)
Get an image's size using only the Python standard library.

`path` is the path of the image file.

`default` is returned if the size could not be ascertained. This usually means the file does not exist or is not in a recognized format. PIL. Only JPG, PNG, GIF, and BMP are supported at this time.

The normal return value is a tuple: `(width, height)`.

The algorithms are based on a PyCode recipe by Perenzo/Welch/Ray.

This helper recognizes fewer image formats and is potentially less accurate than `get_dimensions_pil()`.

Running this module as a script tests this helper. It will print the size of each image file specified on the command line.

CHAPTER

TWELVE

WEBHELPERS.MIMEHELPER

MIME Type helpers

This helper depends on the WebOb package, and has optional Pylons support.

class webhelpers.mimehelper.**MIMETypes**(*environ*)
MIMETypes registration mapping

The MIMETypes object class provides a single point to hold onto all the registered mimetypes, and their association extensions. It's used by the mimetypes method to determine the appropriate content type to return to a client.

classmethod add_alias(*alias*, *mimetype*)
Create a MIMEType alias to a full mimetype.

Examples:

- add_alias('html', 'text/html')
- add_alias('xml', 'application/xml')

An alias may not contain the / character.

classmethod init()
Loads a default mapping of extensions and mimetypes

These are suitable for most web applications by default. Additional types can be added by using the mimetypes module.

mimetype(*content_type*)
Check the PATH_INFO of the current request and client's HTTP Accept to attempt to use the appropriate mime-type.

If a content-type is matched, return the appropriate response content type, and if running under Pylons, set the response content type directly. If a content-type is not matched, return False.

This works best with URLs that end in extensions that differentiate content-type. Examples: http://example.com/example, http://example.com/example.xml, http://example.com/example.csv

Since browsers generally allow for any content-type, but should be sent HTML when possible, the html mimetype check should always come first, as shown in the

example below.

Example:

```
# some code likely in environment.py
MIMETypes.init()
MIMETypes.add_alias('html', 'text/html')
MIMETypes.add_alias('xml', 'application/xml')
MIMETypes.add_alias('csv', 'text/csv')

# code in a Pylons controller
def someaction(self):
    # prepare a bunch of data
    # ......

    # prepare MIMETypes object
    m = MIMETypes(request.environ)

    if m.mimetype('html'):
        return render('/some/template.html')
    elif m.mimetype('atom'):
        return render('/some/xml_template.xml')
    elif m.mimetype('csv'):
        # write the data to a csv file
        return csvfile
    else:
        abort(404)

# Code in a non-Pylons controller.
m = MIMETypes(environ)
response_type = m.mimetype('html')
# ``response_type`` is a MIME type or ``False``.
```

CHAPTER

THIRTEEN

WEBHELPERS.MISC

Helpers that are neither text, numeric, container, or date.

13.1 Data processing

webhelpers.misc.**all**(*seq*[, *pred*])

Is `pred(elm)` true for all elements?

With the default predicate, this is the same as Python 2.5's `all()` function; i.e., it returns true if all elements are true.

```
>>> all(["A", "B"])
True
>>> all(["A", ""])
False
>>> all(["", ""])
False
>>> all(["A", "B", "C"], lambda x: x <= "C")
True
>>> all(["A", "B", "C"], lambda x: x < "C")
False
```

From recipe in itertools docs.

webhelpers.misc.**any**(*seq*[, *pred*])

Is `pred(elm)` is true for any element?

With the default predicate, this is the same as Python 2.5's `any()` function; i.e., it returns true if any element is true.

```
>>> any(["A", "B"])
True
>>> any(["A", ""])
True
>>> any(["", ""])
False
>>> any(["A", "B", "C"], lambda x: x <= "C")
True
```

```
>>> any(["A", "B", "C"], lambda x: x < "C")
True
```

From recipe in itertools docs.

webhelpers.misc.**no**(*seq*[, *pred*])
Is `pred(elm)` false for all elements?

With the default predicate, this returns true if all elements are false.

```
>>> no(["A", "B"])
False
>>> no(["A", ""])
False
>>> no(["", ""])
True
>>> no(["A", "B", "C"], lambda x: x <= "C")
False
>>> no(["X", "Y", "Z"], lambda x: x <="C")
True
```

From recipe in itertools docs.

webhelpers.misc.**count_true**(*seq*[, *pred*])
How many elements is `pred(elm)` true for?

With the default predicate, this counts the number of true elements.

```
>>> count_true([1, 2, 0, "A", ""])
3
>>> count_true([1, "A", 2], lambda x: isinstance(x, int))
2
```

This is equivalent to the `itertools.quantify` recipe, which I couldn't get to work.

webhelpers.misc.**convert_or_none**(*value*, *type_*)
Return the value converted to the type, or None if error.

`type_` may be a Python type or any function taking one argument.

```
>>> print convert_or_none("5", int)
5
>>> print convert_or_none("A", int)
None
```

webhelpers.misc.**flatten**(*iterable*)
Recursively iterate lists and tuples.

Examples:

```
>>> list(flatten([1, [2, 3], 4]))
[1, 2, 3, 4]
>>> list(flatten([1, (2, 3, [4]), 5]))
[1, 2, 3, 4, 5]
```

13.2 Class-related and miscellaneous

class webhelpers.misc.**NotGiven**
A default value for function args.

Use this when you need to distinguish between None and no value.

Example:

```
>>> def foo(arg=NotGiven):
...     print arg is NotGiven
...
>>> foo()
True
>>> foo(None)
False
```

webhelpers.misc.**subclasses_only**(*class_*, *it*, *exclude=None*)
Extract the subclasses of a class from a module, dict, or iterable.

Return a list of subclasses found. The class itself will not be included. This is useful to collect the concrete subclasses of an abstract base class.

class_ is a class.

it is a dict or iterable. If a dict is passed, examine its values, not its keys. To introspect the current module, pass globals(). To introspect another module or namespace, pass vars(the_module_or_namespace).

exclude is an optional list of additional classes to ignore. This is mainly used to exclude abstract subclasses.

13.3 Exceptions and deprecation

webhelpers.misc.**deprecate**(*message*, *pending=False*, *stacklevel=2*)
Issue a deprecation warning.

message: the deprecation message.

pending: if true, use PendingDeprecationWarning. If false (default), use DeprecationWarning. Python displays deprecations and ignores pending deprecations by default.

stacklevel: passed to warnings.warn. The default level 2 makes the traceback end at the caller's level. Higher numbers make it end at higher levels.

webhelpers.misc.**format_exception**(*exc=None*)
Format the exception type and value for display, without the traceback.

This is the function you always wished were in the traceback module but isn't. It's *different* from traceback.format_exception, which includes the traceback, returns a list of lines, and has a trailing newline.

If you don't provide an exception object as an argument, it will call `sys.exc_info()` to get the current exception.

class webhelpers.misc.**DeclarativeException**(*message=None*)

A simpler way to define an exception with a fixed message.

Subclasses have a class attribute `.message`, which is used if no message is passed to the constructor. The default message is the empty string.

Example:

```
>>> class MyException(DeclarativeException):
...     message="can't frob the bar when foo is enabled"
...
>>> try:
...     raise MyException()
... except Exception, e:
...      print e
...
can't frob the bar when foo is enabled
```

class webhelpers.misc.**OverwriteError**(*filename*, *message="not overwriting '%s'"*)

Refusing to overwrite an existing file or directory.

CHAPTER

FOURTEEN

WEBHELPERS.NUMBER

Number formatting, numeric helpers, and numeric statistics.

14.1 Calculations

webhelpers.number.**percent_of**(*part*, *whole*)

What percent of `whole` is `part`?

```
>>> percent_of(5, 100)
5.0
>>> percent_of(13, 26)
50.0
```

14.2 Statistics

webhelpers.number.**mean**(*r*)

Return the mean (i.e., average) of a sequence of numbers.

```
>>> mean([5, 10])
7.5
```

webhelpers.number.**average**(*r*)

Another name for `mean(r)`.

webhelpers.number.**median**(*r*)

Return the median of an iterable of numbers.

The median is the point at which half the numbers are lower than it and half the numbers are higher. This gives a better sense of the majority level than the mean (average) does, because the mean can be skewed by a few extreme numbers at either end. For instance, say you want to calculate the typical household income in a community and you've sampled four households:

```
>>> incomes = [18000]        # Fast food crew
>>> incomes.append(24000)    # Janitor
>>> incomes.append(32000)    # Journeyman
```

```
>>> incomes.append(44000)    # Experienced journeyman
>>> incomes.append(67000)    # Manager
>>> incomes.append(9999999) # Bill Gates
>>> median(incomes)
49500.0
>>> mean(incomes)
1697499.8333333333
```

The median here is somewhat close to the majority of incomes, while the mean is far from anybody's income.

This implementation makes a temporary list of all numbers in memory.

webhelpers.number.**standard_deviation**(*r*, *sample=True*)

Standard deviation.

From the Python Cookbook. Population mode contributed by Lorenzo Catucci.

Standard deviation shows the variability within a sequence of numbers. A small standard deviation means the numbers are close to each other. A large standard deviation shows they are widely different. In fact it shows how far the numbers tend to deviate from the average. This can be used to detect whether the average has been skewed by a few extremely high or extremely low values.

Most natural and random phenomena follow the normal distribution (aka the bell curve), which says that most values are close to average but a few are extreme. E.g., most people are close to 5'9" tall but a few are very tall or very short. If the data does follow the bell curve, 68% of the values will be within 1 standard deviation (stdev) of the average, and 95% will be within 2 standard deviations. So a university professor grading exams on a curve might give a "C" (mediocre) grade to students within 1 stdev of the average score, "B" (better than average) to those within 2 stdevs above, and "A" (perfect) to the 0.25% higher than 2 stdevs. Those between 1 and 2 stdevs below get a "D" (poor), and those below 2 stdevs... we won't talk about them.

By default the helper computes the unbiased estimate for the population standard deviation, by applying an unbiasing factor of sqrt(N/(N-1)).

If you'd rather have the function compute the population standard deviation, pass `sample=False`.

The following examples are taken from Wikipedia. http://en.wikipedia.org/wiki/Standard_deviation

```
>>> standard_deviation([0, 0, 14, 14])
8.082903768654761...
>>> standard_deviation([0, 6, 8, 14])
5.773502691896258...
>>> standard_deviation([6, 6, 8, 8])
1.1547005383792515
>>> standard_deviation([0, 0, 14, 14], sample=False)
7.0
>>> standard_deviation([0, 6, 8, 14], sample=False)
5.0
```

```
>>> standard_deviation([6, 6, 8, 8], sample=False)
1.0
```

(The results reported in Wikipedia are those expected for whole population statistics and therefore are equal to the ones we get by setting `sample=False` in the later tests.)

```
# Fictitious average monthly temperatures in Southern California.
#                       Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
>>> standard_deviation([70, 70, 70, 75, 80, 85, 90, 95, 90, 80, 75, 70])
9.003366373785...
>>> standard_deviation([70, 70, 70, 75, 80, 85, 90, 95, 90, 80, 75, 70], sam
8.620067027323...

# Fictitious average monthly temperatures in Montana.
#                       Jan  Feb  Mar Apr May Jun Jul  Aug Sep Oct Nov Dec
>>> standard_deviation([-32, -10, 20, 30, 60, 90, 100, 80, 60, 30, 10, -32])
45.1378360405574...
>>> standard_deviation([-32, -10, 20, 30, 60, 90, 100, 80, 60, 30, 10, -32],
43.2161878106906...
```

webhelpers.number.**format_number**(*n*, *thousands=', '*, *decimal='.'*)

Format a number with a thousands separator and decimal delimiter.

`n` may be an int, long, float, or numeric string. `thousands` is a separator to put after each thousand. `decimal` is the delimiter to put before the fractional portion if any.

The default style has a thousands comma and decimal point per American usage:

```
>>> format_number(1234567.89)
'1,234,567.89'
>>> format_number(123456)
'123,456'
>>> format_number(-123)
'-123'
```

Various European and international styles are also possible:

```
>>> format_number(1234567.89, " ")
'1 234 567.89'
>>> format_number(1234567.89, " ", ",")
'1 234 567,89'
>>> format_number(1234567.89, ".", ",")
'1.234.567,89'
```

class webhelpers.number.**SimpleStats**(*numeric=False*)

Calculate a few simple statistics on data.

This class calculates the minimum, maximum, and count of all the values given to it. The values are not saved in the object. Usage:

```
>>> stats = SimpleStats()
>>> stats(2)               # Add one data value.
>>> stats.extend([6, 4])   # Add several data values at once.
```

The statistics are available as instance attributes:

```
>>> stats.count
3
>>> stats.min
2
>>> stats.max
6
```

Non-numeric data is also allowed:

```
>>> stats2 = SimpleStats()
>>> stats2("foo")
>>> stats2("bar")
>>> stats2.count
2
>>> stats2.min
'bar'
>>> stats2.max
'foo'
```

`.min` and `.max` are `None` until the first data value is registered.

Subclasses can override `._init_stats` and `._update_stats` to add additional statistics.

The constructor accepts one optional argument, `numeric`. If true, the instance accepts only values that are `int`, `long`, or `float`. The default is false, which accepts any value. This is meant for instances or subclasses that don't want non-numeric values.

__call__(*value*)
Add a data value.

extend(*values*)
Add several data values at once, akin to `list.extend`.

class `webhelpers.number.`**Stats**
A container for data and statistics.

This class extends `SimpleStats` by calculating additional statistics, and by storing all data seen. All values must be numeric (`int`, `long`, and/or `float`), and you must call `.finish()` to generate the additional statistics. That's because the statistics here cannot be calculated incrementally, but only after all data is known.

```
>>> stats = Stats()
>>> stats.extend([5, 10, 10])
>>> stats.count
3
>>> stats.finish()
>>> stats.mean
8.33333333333333...
>>> stats.median
10
>>> stats.standard_deviation
```

```
2.8867513459481287
```

All data is stored in a list and a set for later use:

```
>>> stats.list
[5, 10, 10]

>>  stats.set
set([5, 10])
```

(The double prompt ">>" is used to hide the example from doctest.)

The stat attributes are `None` until you call `.finish()`. It's permissible – though not recommended – to add data after calling `.finish()` and then call `.finish()` again. This recalculates the stats over the entire data set.

In addition to the hook methods provided by `SimpleStats`, subclasses can override `._finish-stats` to provide additional statistics.

__call__(*value*)
Add a data value.

extend(*values*)
Add several data values at once, akin to `list.extend`.

finish()
Finish calculations. (Call after adding all data values.)

Call this after adding all data values, or the results will be incomplete.

14.3 Number formatting

webhelpers.number.**format_data_size**(*size*, *unit*, *precision=1*, *binary=False*, *full_name=False*)
Format a number using SI units (kilo, mega, etc.).

`size`: The number as a float or int.

`unit`: The unit name in plural form. Examples: "bytes", "B".

`precision`: How many digits to the right of the decimal point. Default is 1. 0 suppresses the decimal point.

`binary`: If false, use base-10 decimal prefixes (kilo = K = 1000). If true, use base-2 binary prefixes (kibi = Ki = 1024).

`full_name`: If false (default), use the prefix abbreviation ("k" or "Ki"). If true, use the full prefix ("kilo" or "kibi"). If false, use abbreviation ("k" or "Ki").

Examples:

```
>>> format_data_size(1024, "B")
'1.0 kB'
>>> format_data_size(1024, "B", 2)
```

```
'1.02 kB'
>>> format_data_size(1024, "B", 2, binary=True)
'1.00 KiB'
>>> format_data_size(54000, "Wh", 0)
'54 kWh'
>>> format_data_size(85000, "m/h", 0)
'85 km/h'
>>> format_data_size(85000, "m/h", 0).replace("km/h", "klicks")
'85 klicks'
```

webhelpers.number.**format_byte_size**(*size*, *precision=1*, *binary=False*, *full_name=False*)

Same as `format_data_size` but specifically for bytes.

Examples:

```
>>> format_byte_size(2048)
'2.0 kB'
>>> format_byte_size(2048, full_name=True)
'2.0 kilobytes'
```

webhelpers.number.**format_bit_size**(*size*, *precision=1*, *binary=False*, *full_name=False*)

Same as `format_data_size` but specifically for bits.

Examples:

```
>>> format_bit_size(2048)
'2.0 kb'
>>> format_bit_size(2048, full_name=True)
'2.0 kilobits'
```

CHAPTER

FIFTEEN

WEBHELPERS.PAGINATE

15.1 Page Objects

15.2 URL generators

CHAPTER

SIXTEEN

WEBHELPERS.TEXT

CHAPTER

SEVENTEEN

WEBHELPERS.UTIL

Utility functions used by various web helpers.

This module contains support functions used by other helpers, and functions for URL manipulation. Most of these helpers predate the 0.6 reorganization; they would have been put in other subpackages if they have been created later.

webhelpers.util.**update_params**(*_url*, *_debug=False*, ***params*)

Update query parameters in a URL.

`_url` is any URL, with or without a query string.

`\*\*params` are query parameters to add or replace. Each value may be a string, a list of strings, or None. Passing a list generates multiple values for the same parameter. Passing None deletes the corresponding parameter if present.

Return the new URL.

Debug mode: if a pseudo-parameter `_debug=True` is passed, return a tuple: `[0]` is the URL without query string or fragment, `[1]` is the final query parameters as a dict, and `[2]` is the fragment part of the original URL or the empty string.

Usage:

```
>>> update_params("foo", new1="NEW1")
'foo?new1=NEW1'
>>> update_params("foo?p=1", p="2")
'foo?p=2'
>>> update_params("foo?p=1", p=None)
'foo'
>>> update_params("http://example.com/foo?new1=OLD1#myfrag", new1="NEW1")
'http://example.com/foo?new1=NEW1#myfrag'
>>> update_params("http://example.com/foo?new1=OLD1#myfrag", new1="NEW1", _
('http://example.com/foo', {'new1': 'NEW1'}, 'myfrag')
>>> update_params("http://www.mau.de?foo=2", brrr=3)
'http://www.mau.de?foo=2&brrr=3'
>>> update_params("http://www.mau.de?foo=A&foo=B", foo=["C", "D"])
'http://www.mau.de?foo=C&foo=D'
```

webhelpers.util.**cgi_escape**(*s*, *quote=False*)

Replace special characters '&', '<' and '>' by SGML entities.

This is a slightly more efficient version of the cgi.escape by using 'in' membership to test if the replace is needed.

This function returns a plain string. Programs using the HTML builder should call `webhelpers.html.builder.escape()` instead of this to prevent double-escaping.

Changed in WebHelpers 1.2: escape single-quote as well as double-quote.

`webhelpers.util.`**`html_escape`**(*s*)
HTML-escape a string or object.

This converts any non-string objects passed into it to strings (actually, using `unicode()`). All values returned are non-unicode strings (using `&#num;` entities for all non-ASCII characters).

None is treated specially, and returns the empty string.

This function returns a plain string. Programs using the HTML builder should wrap the result in `literal()` to prevent double-escaping.

`webhelpers.util.`**`iri_to_uri`**(*iri*)
Convert an IRI portion to a URI portion suitable for inclusion in a URL.

(An IRI is an Internationalized Resource Identifier.)

This is the algorithm from section 3.1 of RFC 3987. However, since we are assuming input is either UTF-8 or unicode already, we can simplify things a little from the full method.

Returns an ASCII string containing the encoded result.

class `webhelpers.util.`**`Partial`**(**args*, ***kw*)
A partial function object.

Equivalent to functools.partial, which was introduced in Python 2.5.

class `webhelpers.util.`**`SimplerXMLGenerator`**(*out=None*, *encoding='iso-8859-1'*)
A subclass of Python's SAX XMLGenerator.

`addQuickElement`(*name*, *contents=None*, *attrs=None*)
Add an element with no children.

class `webhelpers.util.`**`UnicodeMultiDict`**(*multi=None*, *encoding=None*, *errors='strict'*, *decode_keys=False*)
A MultiDict wrapper that decodes returned values to unicode on the fly.

Decoding is not applied to assigned values.

The key/value contents are assumed to be `str`/`strs` or `str`/`FieldStorages` (as is returned by the `paste.request.parse()` functions).

Can optionally also decode keys when the `decode_keys` argument is True.

`FieldStorage` instances are cloned, and the clone's `filename` variable is decoded. Its `name` variable is decoded when `decode_keys` is enabled.

add(*key*, *value*)
Add the key and value, not overwriting any previous value.

clear()

copy()

dict_of_lists()
Return dict where each key is associated with a list of values.

getall(*key*)
Return list of all values matching the key (may be an empty list).

getone(*key*)
Return one value matching key. Raise KeyError if multiple matches.

has_key(*key*)

items()

iteritems()

iterkeys()

itervalues()

keys()

mixed()
Return dict where values are single values or a list of values.

The value is a single value if key appears just once. It is a list of values when a key/value appears more than once in this dictionary. This is similar to the kind of dictionary often used to represent the variables in a web request.

pop(*key*, **args*)

popitem()

setdefault(*key*, *default=None*)

values()

CHAPTER

EIGHTEEN

PYLONS-SPECIFIC SUBPACKAGES

These work ONLY with the Pylons web framework and its derivatives (TurboGears 2). They are NOT compatible with Pyramid; see the submodule pages for alternatives.

18.1 webhelpers.pylonslib

18.2 webhelpers.pylonslib.flash

18.2.1 Classes

18.3 webhelpers.pylonslib.grid

18.4 webhelpers.pylonslib.minify

18.5 webhelpers.pylonslib.secure_form

CHAPTER

NINETEEN

NON-ESSENTIAL SUBPACKAGES

19.1 webhelpers.markdown

`webhelpers.markdown` is a copy of Markdown 1.7, used as a fallback for `webhelpers.html.converters.markdown()` if the full Markdown package is not installed. See the Markdown website for documentation on the Markdown format and this module. Markdown is now at version 2.x and contains new features and plugins which are too big to include in WebHelpers. There is also an alternate implementation called Markdown2. Both are available on PyPI. See the `markdown()` documentation for how to use them with WebHelpers.

19.2 webhelpers.textile

`webhelpers.textile` is a copy of Textile, used by `webhelpers.html.converters.textilize()`. See the Textile site for documentation on the Textile format and this module.

PYTHON MODULE INDEX

W

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T