WebHelpers Documentation

Release 1.3

Ben Bangert

June 16, 2016

Contents

All helpers are organized into subpackages.

CHAPTER 1

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 2

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", "honey"]
>>> 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"}, ["To", "From"])
    >>> 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 3

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 4

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/journalism/publishing.",
...     language=u"en",
... )
>>> feed.add_item(title="Hello", link=u"http://www.holovaty.com/test/", description="Testing.")
>>> 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*)

mime_type = 'application/rss+xml'

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*)

mime_type = 'application/atom+xml'

ns = u'http://www.w3.org/2005/Atom'

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 5

webhelpers.html

HTML generation helpers.

All public objects in the `webhelpers.html.builder` subpackage are also available in the `webhelpers.html` namespace. Most programs will want to put this line in their code:

```
from webhelpers.html import *
```

Or you can import the most frequently-used objects explicitly:

```
from webhelpers.html import HTML, escape, literal
```

CHAPTER 6

webhelpers.html.builder

HTML/XHTML tag builder

HTML Builder provides:

- an `HTML` object that creates (X)HTML tags in a Pythonic way.
- a `literal` class used to mark strings containing intentional HTML markup.
- a smart `escape()` function that preserves literals but escapes other strings that may accidentally contain markup characters ("<", ">", "&", "", "") or malicious Javascript tags. Escaped strings are returned as literals to prevent them from being double-escaped later.

`literal` is a subclass of `unicode`, so it works with all string methods and expressions. The only thing special about it is the `.__html__` method, which returns the string itself. The `escape()` function follows a simple protocol: if the object has an `.__html__` method, it calls that rather than `.__str__` to get the HTML representation. Third-party libraries that do not want to import `literal` (and this create a dependency on WebHelpers) can put an `.__html__` method in their own classes returning the desired HTML representation.

WebHelpers 1.2 uses MarkupSafe, a package which provides an enhanced implementation of this protocol. Mako and Pylons have also switched to MarkupSafe. Its advantages are a C speedup for escaping, escaping single-quotes for security, and adding new methods to `literal`. **literal** is now a subclass of `markupsafe.Markup`. **escape** is `markupsafe.escape_silent`. (The latter does not exist yet in MarkupSafe 0.9.3, but WebHelpers itself converts None to "" in the meantime).

Single-quote escaping affects HTML attributes that are written like this: *alt='Some text.'* rather than the normal *alt="Some text."* If the text is a replaceable parameter whose value contains a single quote, the browser would think the value ends earlier than it does, thus enabling a potential cross-site scripting (XSS) attack. WebHelpers 1.0 and earlier escaped double quotes but not single quotes. MarkupSafe escapes both double and single quotes, preventing this sort of attack.

MarkupSafe has some slight differences which should not cause compatibility issues but may in the following edge cases. (A) The `force` argument to `escape()` is gone. We doubt it was ever used. (B) The default encoding of `literal()` is "ascii" instead of "utf-8". (C) Double quotes are escaped as """ instead of """. Single quotes are escaped as "'".

When `literal` is used in a mixed expression containing both literals and ordinary strings, it tries hard to escape the strings and return a literal. However, this depends on which value has "control" of the expression. `literal` seems to be able to take control with all combinations of the + operator, but with `%` and `join` it must be on the left side of the expression. So these all work:

```
"A" + literal("B")
literal(", ").join(["A", literal("B")])
literal("%s %s") % (16, literal("kg"))
```

But these return an ordinary string which is prone to double-escaping later:

```
"\n".join([literal('<span class="foo">Foo!</span>'), literal('Bar!')])
"%s %s" % (literal("16"), literal("&lt;em&gt;kg&lt;/em&gt;"))
```

Third-party libraries that don't want to import `literal` and thus avoid a dependency on WebHelpers can add an `.__html__` method to any class, which can return the same as `.__str__` or something else. `escape()` trusts the HTML method and does not escape the return value. So only strings that lack an `.__html__` method will be escaped.

The `HTML` object has the following methods for tag building:

`HTML(*strings)` Escape the string args, concatenate them, and return a literal. This is the same as `escape(s)` but accepts multiple strings. Multiple args are useful when mixing child tags with text, such as:

```
html = HTML("The king is a >>", HTML.strong("fink"), "<<!")
```

`HTML.literal(*strings)` Same as `literal` but concatenates multiple arguments.

`HTML.comment(*strings)` Escape and concatenate the strings, and wrap the result in an HTML comment.

`HTML.tag(tag, *content, **attrs)` Create an HTML tag `tag` with the keyword args converted to attributes. The other positional args become the content for the tag, and are escaped and concatenated. If an attribute name conflicts with a Python keyword (notably "class"), append an underscore. If an attribute value is `None`, the attribute is not inserted. Two special keyword args are recognized:

`c` Specifies the content. This cannot be combined with content in positional args. The purpose of this argument is to position the content at the end of the argument list to match the native HTML syntax more closely. Its use is entirely optional. The value can be a string, a tuple, or a tag.

`_closed` If present and false, do not close the tag. Otherwise the tag will be closed with a closing tag or an XHTML-style trailing slash as described below.

`_nl` If present and true, insert a newline before the first content element, between each content element, and at the end of the tag.

Example:

```
>>> HTML.tag("a", href="http://www.yahoo.com", name=None,
... c="Click Here")
literal(u'<a href="http://www.yahoo.com">Click Here</a>')
```

`HTML.__getattr__` Same as `HTML.tag` but using attribute access. Example:

```
>>> HTML.a("Foo", href="http://example.com/", class_="important")
literal(u'<a class="important" href="http://example.com/">Foo</a>')
```

`HTML.cdata` Wrap the text in a "<![CDATA[...]]>" section. Plain strings will not be escaped because CDATA itself is an escaping syntax.

```
>>> HTML.cdata(u"Foo")
literal(u'<![CDATA[Foo]]>')
```

```
>>> HTML.cdata(u"<p>")
literal(u'<![CDATA[<p>]]>')
```

6.1 About XHTML and HTML

This builder always produces tags that are valid as *both* HTML and XHTML. "Void" tags – those which can never have content like `<br>` and `<input>` – are written like `<br />`, with a space and a trailing `/`.

Only void tags get this treatment. The library will never, for example, produce `<script src="..." />`, which is invalid HTML. Instead it will produce `<script src="..."></script>`.

The W3C HTML validator validates these constructs as valid HTML Strict. It does produce warnings, but those warnings warn about the ambiguity if this same XML-style self-closing tags are used for HTML elements that are allowed to take content (`<script>`, `<textarea>`, etc). This library never produces markup like that.

Rather than add options to generate different kinds of behavior, we felt it was better to create markup that could be used in different contexts without any real problems and without the overhead of passing options around or maintaining different contexts, where you'd have to keep track of whether markup is being rendered in an HTML or XHTML context.

If you _really_ want tags without training slashes (e.g., `<br>`), you can abuse ``_closed=False` to produce them.

6.2 Classes

class webhelpers.html.builder.**literal**(*s*, *encoding=None*, *errors='strict'*)

Represents an HTML literal.

This subclass of unicode has a `.__html__()` method that is detected by the `escape()` function.

Also, if you add another string to this string, the other string will be quoted and you will get back another literal object. Also `literal(...) % obj` will quote any value(s) from `obj`. If you do something like `literal(...) + literal(...)`, neither string will be changed because `escape(literal(...))` doesn't change the original literal.

Changed in WebHelpers 1.2: the implementation is now now a subclass of `markupsafe.Markup`. This brings some new methods: `.escape` (class method), `.unescape`, and `.striptags`.

classmethod escape(*s*)

Same as the `escape` function but return the proper subclass in subclasses.

unescape()

Unescape markup again into an text_type string. This also resolves known HTML4 and XHTML entities:

```
>>> Markup("Main &raquo; <em>About</em>").unescape()
u'Main \xbb <em>About</em>'
```

striptags()

Unescape markup into an text_type string and strip all tags. This also resolves known HTML4 and XHTML entities. Whitespace is normalized to one:

```
>>> Markup("Main &raquo;  <em>About</em>").striptags()
u'Main \xbb About'
```

class webhelpers.html.builder.**HTML**

Described above.

6.3 Functions

webhelpers.html.builder.**lit_sub**(**args*, ***kw*)

Literal-safe version of re.sub. If the string to be operated on is a literal, return a literal result. All arguments are passed directly to `re.sub`.

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 7

webhelpers.html.converters

Functions that convert from text markup languages to HTML and back.

webhelpers.html.converters.**format_paragraphs**(*text*, *preserve_lines=False*)
Convert text to HTML paragraphs.

text: the text to convert. Split into paragraphs at blank lines (i.e., wherever two or more consecutive newlines appear), and wrap each paragraph in a <p>.

preserve_lines: If true, add
 before each single line break

webhelpers.html.converters.**markdown**(*text*, *markdown=None*, ***kwargs*)
Format the text to HTML with Markdown formatting.

Markdown is a wiki-like text markup language, originally written by John Gruber for Perl. The helper converts Markdown text to HTML.

There are at least two Python implementations of Markdown. Markdown <http://www.freewisdom.org/projects/python-markdown/>‘_is the original port, and version 2.x contains extensions for footnotes, RSS, etc. Markdown2 is another port which claims to be faster and to handle edge cases better.

You can pass the desired Markdown module as the `markdown` argument, or the helper will try to import `markdown`. If neither is available, it will fall back to `webhelpers.markdown`, which is Freewisdom's Markdown 1.7 without extensions.

IMPORTANT: If your source text is untrusted and may contain malicious HTML markup, pass `safe_mode="escape"` to escape it, `safe_mode="replace"` to replace it with a scolding message, or `safe_mode="remove"` to strip it.

webhelpers.html.converters.**nl2br**(*text*)
Insert a
 before each newline.

webhelpers.html.converters.**textilize**(*text*, *sanitize=False*)
Format the text to HTML with Textile formatting.

This function uses the PyTextile library which is included with WebHelpers.

Additionally, the output can be sanitized which will fix tags like <img />,
 and <hr /> for proper XHTML output.

CHAPTER 8

webhelpers.html.grid

A helper to make an HTML table from a list of dicts, objects, or sequences.

A set of CSS styles complementing this helper is in "webhelpers/html/public/stylesheets/grid.css". To use them, include the stylesheet in your applcation and set your <table> class to "stylized".

The documentation below is not very clear. This is a known bug. We need a native English speaker who uses the module to volunteer to rewrite it.

This module is written and maintained by Ergo^.

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

class webhelpers.html.grid.**Grid**(*itemlist*, *columns*, *column_labels=None*, *column_formats=None*, *start_number=1*, *order_column=None*, *order_direction=None*, *request=None*, *url=None*, ***kw*)

This class is designed to aid programmer in the task of creation of tables/grids - structures that are mostly built from datasets.

To create a grid at minimum one one needs to pass a dataset, like a list of dictionaries, or sqlalchemy proxy or query object:

```
grid = Grid(itemlist, ['_numbered','c1', 'c2','c4'])
```

where itemlist in this simple scenario is a list of dicts:

[{'c1':1,'c2'...}, {'c1'...}, ...]

This helper also received the list that defines order in which columns will be rendered - also keep note of special column name that can be passed in list that defines order - `_numbered` - this adds additional column that shows the number of item. For paging sql data there one can pass `start_number` argument to the grid to define where to start counting. Descendant sorting on `_numbered` column decrements the value, you can change how numberign function behaves by overloading `calc_row_no` property.

Converting the grid to a string renders the table rows. That's *just* the <tr> tags, not the <table> around them. The part outside the <tr>s have too many variations for us to render it. In many template systems, you can simply assign the grid to a template variable and it will be automatically converted to a string. Example using a Mako template:

```
<table class="stylized">
<caption>My Lovely Grid</caption>
<col class="c1" />
${my_grid}
</table>
```

The names of the columns will get automatically converted for humans ie. foo_bar becomes Foo Bar. If you want the title to be something else you can change the grid.labels dict. If you want the column `part_no` to become `Catalogue Number` just do:

```
grid.labels[``part_no``] = u'Catalogue Number'
```

It may be desired to exclude some or all columns from generation sorting urls (used by subclasses that are sorting aware). You can use grids exclude_ordering property to pass list of columns that should not support sorting. By default sorting is disabled - this `exclude_ordering` contains every column name.

Since various programmers have different needs, Grid is highly customizable. By default grid attempts to read the value from dict directly by key. For every column it will try to output value of current_row['colname'].

Since very often this behavior needs to be overridden like we need date formatted, use conditionals or generate a link one can use the `column_formats` dict and pass a rendering function/lambda to it. For example we want to apppend `foo` to part number:

```
def custem_part_no_td(col_num, i, item):
    return HTML.td(`Foo %s` % item[``part_no``])

grid.column_formats[``part_no``] = custem_part_no_td
```

You can customize the grids look and behavior by overloading grids instance render functions:

```
grid.default_column_format(self, column_number, i, record, column_name)
by default generates markup like:
<td class="cNO">VALUE</td>

grid.default_header_column_format(self, column_number, column_name,
    header_label)
by default generates markup like:
<td class="cNO COLUMN_NAME">VALUE</td>

grid.default_header_ordered_column_format(self, column_number, order,
    column_name, header_label)
Used by grids that support ordering of columns in the grid like,
webhelpers.pylonslib.grid.GridPylons.
by default generates markup like:
<td class="cNO ordering ORDER_DIRECTION COLUMN_NAME">LABEL</td>

grid.default_header_record_format(self, headers)
by default generates markup like:
<tr class="header">HEADERS_MARKUP</tr>

grid.default_record_format(self, i, record, columns)
Make an HTML table from a list of objects, and soon a list of
sequences, a list of dicts, and a single dict.
<tr class="ODD_OR_EVEN">RECORD_MARKUP</tr>

grid.generate_header_link(self, column_number, column, label_text)
by default just sets the order direction and column properties for grid.
Actual link generation is handled by sublasses of Grid.
```

```
    grid.numbered_column_format(self, column_number, i, record)
    by default generates markup like:
    <td class="cNO">RECORD_NO</td>
```

generate_header_link(*column_number*, *column*, *label_text*)

This handles generation of link and then decides to call `self.default_header_ordered_column_format` or `self.default_header_column_format` based on whether current column is the one that is used for sorting.

you need to extend Grid class and overload this method implementing ordering here, whole operation consists of setting self.order_column and self.order_dir to their CURRENT values, and generating new urls for state that header should set set after its clicked

(additional kw are passed to url gen. - like for webhelpers.paginate) example URL generation code below:

```
        GET = dict(self.request.copy().GET) # needs dict() for py2.5 compat
        self.order_column = GET.pop("order_col", None)
        self.order_dir = GET.pop("order_dir", None)
        # determine new order
        if column == self.order_column and self.order_dir == "asc":
            new_order_dir = "dsc"
        else:
            new_order_dir = "asc"
        self.additional_kw['order_col'] = column
        self.additional_kw['order_dir'] = new_order_dir
        # generate new url for example url_generator uses
        # pylons's url.current() or pyramid's current_route_url()
        new_url = self.url_generator(**self.additional_kw)
        # set label for header with link
        label_text = HTML.tag("a", href=new_url, c=label_text)
```

class webhelpers.html.grid.**ObjectGrid**(*itemlist*, *columns*, *column_labels=None*, *column_formats=None*, *start_number=1*, *order_column=None*, *order_direction=None*, *request=None*, *url=None*, ***kw*)

A grid class for a sequence of objects.

This grid class assumes that the rows are objects rather than dicts, and uses attribute access to retrieve the column values. It works well with SQLAlchemy ORM instances.

CHAPTER 9

webhelpers.html.tags

Helpers that produce simple HTML tags.

Most helpers have an `**attrs` argument to specify additional HTML attributes. A trailing underscore in the name will be deleted; this is especially important for attributes that are identical to Python keywords; e.g., `class_`. Some helpers handle certain keywords specially; these are noted in the helpers' docstrings.

To create your own custom tags, see `webhelpers.html.builder`.

A set of CSS styles complementing these helpers is in `webhelpers/public/stylesheets/webhelpers.css`.

9.1 Form tags

webhelpers.html.tags.**form**(*url*, *method='post'*, *multipart=False*, *hidden_fields=None*, ***attrs*)

An open tag for a form that will submit to `url`.

You must close the form yourself by calling `end_form()` or outputting </form>.

Options:

method The method to use when submitting the form, usually either "GET" or "POST". If "PUT", "DELETE", or another verb is used, a hidden input with name _method is added to simulate the verb over POST.

multipart If set to True, the enctype is set to "multipart/form-data". You must set it to true when uploading files, or the browser will submit the filename rather than the file.

hidden_fields Additional hidden fields to add to the beginning of the form. It may be a dict or an iterable of key-value tuples. This is implemented by calling the object's `.items()` method if it has one, or just iterating the object. (This will successfuly get multiple values for the same key in WebOb MultiDict objects.)

Because input tags must be placed in a block tag rather than directly inside the form, all hidden fields will be put in a ' '. The style prevents the from being displayed or affecting the layout.

Examples:

```
>>> form("/submit")
literal(u'<form action="/submit" method="post">')
>>> form("/submit", method="get")
literal(u'<form action="/submit" method="get">')
>>> form("/submit", method="put")
literal(u'<form action="/submit" method="post"><div style="display:none">\n<input name="_method"
```

```
>>> form("/submit", "post", multipart=True)
literal(u'<form action="/submit" enctype="multipart/form-data" method="post">')
```

Changed in WebHelpers 1.0b2: add and `hidden_fields` arg.

Changed in WebHelpers 1.2: don't add an "id" attribute to hidden tags generated by this helper; they clash if there are multiple forms on the page.

`webhelpers.html.tags.`**`end_form`**`()`
Output "</form>".

Example:

```
>>> end_form()
literal(u'</form>')
```

`webhelpers.html.tags.`**`text`**`(`*name*, *value=None*, *id=<class 'webhelpers.misc.NotGiven'>*, *type='text'*, ***attrs*`)`
Create a standard text field.

`value` is a string, the content of the text field.

`id` is the HTML ID attribute, and should be passed as a keyword argument. By default the ID is the same as the name filtered through `_make_safe_id_component()`. Pass None to suppress the ID attribute entirely.

`type` is the input field type, normally "text". You can override it for HTML 5 input fields that don't have their own helper; e.g., "search", "email", "date".

Options:

- **`disabled` - If set to True, the user will not be able to use** this input.
- **`size` - The number of visible characters that will fit in the** input.
- **`maxlength` - The maximum number of characters that the browser** will allow the user to enter.

The remaining keyword args will be standard HTML attributes for the tag.

Example, a text input field:

```
>>> text("address")
literal(u'<input id="address" name="address" type="text" />')
```

HTML 5 example, a color picker:

```
>>> text("color", type="color")
literal(u'<input id="color" name="color" type="color" />')
```

`webhelpers.html.tags.`**`textarea`**`(`*name*, *content=''*, *id=<class 'webhelpers.misc.NotGiven'>*, ***attrs*`)`
Create a text input area.

Example:

```
>>> textarea("body", "", cols=25, rows=10)
literal(u'<textarea cols="25" id="body" name="body" rows="10"></textarea>')
```

`webhelpers.html.tags.`**`hidden`**`(`*name*, *value=None*, *id=<class 'webhelpers.misc.NotGiven'>*, ***attrs*`)`
Create a hidden field.

`webhelpers.html.tags.`**`file`**`(`*name*, *value=None*, *id=<class 'webhelpers.misc.NotGiven'>*, ***attrs*`)`
Create a file upload field.

If you are using file uploads then you will also need to set the multipart option for the form.

Example:

```
>>> file('myfile')
literal(u'<input id="myfile" name="myfile" type="file" />')
```

webhelpers.html.tags.**password**(*name*, *value=None*, *id=<class 'webhelpers.misc.NotGiven'>*, ***attrs*)

Create a password field.

Takes the same options as `text()`.

webhelpers.html.tags.**checkbox**(*name*, *value='1'*, *checked=False*, *label=None*, *id=<class 'webhelpers.misc.NotGiven'>*, ***attrs*)

Create a check box.

Arguments: `name` – the widget's name.

`value` – the value to return to the application if the box is checked.

`checked` – true if the box should be initially checked.

`label` – a text label to display to the right of the box.

`id` is the HTML ID attribute, and should be passed as a keyword argument. By default the ID is the same as the name filtered through `_make_safe_id_component()`. Pass None to suppress the ID attribute entirely.

The following HTML attributes may be set by keyword argument:

- `disabled` - If true, checkbox will be grayed out.
- `readonly` - If true, the user will not be able to modify the checkbox.

To arrange multiple checkboxes in a group, see webhelpers.containers.distribute().

Example:

```
>>> checkbox("hi")
literal(u'<input id="hi" name="hi" type="checkbox" value="1" />')
```

webhelpers.html.tags.**radio**(*name*, *value*, *checked=False*, *label=None*, ***attrs*)

Create a radio button.

Arguments: `name` – the field's name.

`value` – the value returned to the application if the button is pressed.

`checked` – true if the button should be initially pressed.

`label` – a text label to display to the right of the button.

The id of the radio button will be set to the name + '_' + value to ensure its uniqueness. An `id` keyword arg overrides this. (Note that this behavior is unique to the `radio()` helper.)

To arrange multiple radio buttons in a group, see webhelpers.containers.distribute().

webhelpers.html.tags.**submit**(*name*, *value*, *id=<class 'webhelpers.misc.NotGiven'>*, ***attrs*)

Create a submit button with the text `value` as the caption.

webhelpers.html.tags.**select**(*name*, *selected_values*, *options*, *id=<class 'webhelpers.misc.NotGiven'>*, ***attrs*)

Create a dropdown selection box.

- `name` – the name of this control.
- `selected_values` – a string or list of strings or integers giving the value(s) that should be preselected.

- `options` – an `Options` object or iterable of `(value, label)` pairs. The label will be shown on the form; the option will be returned to the application if that option is chosen. If you pass a string or int instead of a 2-tuple, it will be used for both the value and the label. If the *value* is a tuple or a list, it will be added as an optgroup, with *label* as label.

`id` is the HTML ID attribute, and should be passed as a keyword argument. By default the ID is the same as the name. filtered through `_make_safe_id_component()`. Pass None to suppress the ID attribute entirely.

CAUTION: the old rails helper `options_for_select` had the label first. The order was reversed because most real-life collections have the value first, including dicts of the form `{value: label}`. For those dicts you can simply pass `D.items()` as this argument.

HINT: You can sort options alphabetically by label via: `sorted(my_options, key=lambda x: x[1])`

The following options may only be keyword arguments:

- **`multiple` – if true, this control will allow multiple** selections.
- `prompt` – if specified, an extra option will be prepended to the list: (“”, `prompt`). This is intended for those “Please choose ...” pseudo-options. Its value is “”, equivalent to not making a selection.

Any other keyword args will become HTML attributes for the <select>.

Examples (call, result):

```
>>> select("currency", "$", [["$", "Dollar"], ["DKK", "Kroner"]])
literal(u'<select id="currency" name="currency">\n<option selected="selected" value="$">Dollar</
>>> select("cc", "MasterCard", [ "VISA", "MasterCard" ], id="cc", class_="blue")
literal(u'<select class="blue" id="cc" name="cc">\n<option value="VISA">VISA</option>\n<option s
>>> select("cc", ["VISA", "Discover"], [ "VISA", "MasterCard", "Discover" ])
literal(u'<select id="cc" name="cc">\n<option selected="selected" value="VISA">VISA</option>\n<o
>>> select("currency", None, [["$", "Dollar"], ["DKK", "Kroner"]], prompt="Please choose ...")
literal(u'<select id="currency" name="currency">\n<option selected="selected" value="">Please ch
>>> select("privacy", 3L, [(1, "Private"), (2, "Semi-public"), (3, "Public")])
literal(u'<select id="privacy" name="privacy">\n<option value="1">Private</option>\n<option valu
>>> select("recipients", None, [(["u1", "User1"), ("u2", "User2")], "Users"), ([("g1", "Group1"
literal(u'<select id="recipients" name="recipients">\n<optgroup label="Users">\n<option value="u
```

class `webhelpers.html.tags.`**`Options`**

A tuple of `Option` objects for the `select()` helper.

`select()` calls this automatically so you don’t have to. However, you may find it useful for organizing your code, and its methods can be convenient.

This class has multiple jobs:

- Canonicalize the options given to `select()` into a consistent format.
- Avoid canonicalizing the same data multiple times. It subclasses tuple rather than a list to guarantee that nonconformant elements won’t be added after canonicalization.
- Provide convenience methods to iterate the values and labels separately.

```
>>> opts = Options(["A", 1, ("b", "B")])
>>> opts
Options([(u'A', u'A'), (u'1', u'1'), (u'b', u'B')])
>>> list(opts.values())
[u'A', u'1', u'b']
>>> list(opts.labels())
[u'A', u'1', u'B']
>>> opts[2].value
u'b'
```

```
>>> opts[2].label
u'B'
```

labels()
Iterate the label element of each pair.

values()
Iterate the value element of each pair.

class webhelpers.html.tags.**Option**(*value*, *label*)
An option for an HTML select.

A simple container with two attributes, `.value` and `.label`.

class webhelpers.html.tags.**OptGroup**(*label*, *options*)
A container for Options

webhelpers.html.tags.**title**(*title*, *required=False*, *label_for=None*)
Format the user-visible title for a form field.

Use this for forms that have a text title above or next to each field.

`title` – the name of the field; e.g., "First Name".

`required` – if true, append a *" to the title and use the 'required' HTML format (see example); otherwise use the 'not required' format.

`label_for` – if provided, put `<label for="ID">` around the title. The value should be the HTML ID of the input field related to this title. Per the HTML standard, the ID should point to a single control (input, select, textarea), not to multiple controls (fieldset, group of checkboxes, group of radio buttons). ID's are set by passing the keyword arg `id` to the appropriate helper.

Note that checkboxes and radio buttions typically have their own individual labels in addition to the title. You can set these with the `label` argument to `checkbox()` and `radio()`.

This helper does not accept other keyword arguments.

See webhepers/public/stylesheets/webhelpers.css for suggested styles.

```
>>> title("First Name")
literal(u'<span class="not-required">First Name</span>')
>>> title("Last Name", True)
literal(u'<span class="required">Last Name <span class="required-symbol">*</span></span>')
>>> title("First Name", False, "fname")
literal(u'<span class="not-required"><label for="fname">First Name</label></span>')
>>> title("Last Name", True, label_for="lname")
literal(u'<span class="required"><label for="lname">Last Name</label> <span class="required-symb
```

webhelpers.html.tags.**required_legend**()
Return an inline HTML snippet explaining which fields are required.

See webhepers/public/stylesheets/webhelpers.css for suggested styles.

```
>>> required_legend()
literal(u'<span class="required required-symbol">*</span> = required')
```

9.2 ModelTags class

class webhelpers.html.tags.**ModelTags**(*record*, *use_keys=False*, *date_format='%m/%d/%Y'*, *id_format=None*)

A nice way to build a form for a database record.

ModelTags allows you to build a create/update form easily. (This is the C and U in CRUD.) The constructor takes a database record, which can be a SQLAlchemy mapped class, or any object with attributes or keys for the field values. Its methods shadow the the form field helpers, but it automatically fills in the value attribute based on the current value in the record. (It also knows about the 'checked' and 'selected' attributes for certain tags.)

You can also use the same form to input a new record. Pass `None` or `""` instead of a record, and it will set all the current values to a default value, which is either the *default* keyword arg to the method, or *""* if not specified.

(Hint: in Pylons you can put `mt = ModelTags(c.record)` in your template, and then if the record doesn't exist you can either set `c.record = None` or not set it at all. That's because nonexistent `c` attributes resolve to *""* unless you've set `config["pylons.strict_c"] = True`. However, having a `c` attribute that's sometimes set and sometimes not is arguably bad programming style.)

checkbox(*name*, *value='1'*, *label=None*, ***kw*)

Build a checkbox field.

The box will be initially checked if the value of the corresponding database field is true.

The submitted form value will be "1" if the box was checked. If the box is unchecked, no value will be submitted. (This is a downside of the standard checkbox tag.)

To display multiple checkboxes in a group, see webhelper.containers.distribute().

date(*name*, ***kw*)

Same as text but format a date value into a date string.

The value can be a *datetime.date*, *datetime.datetime*, *None*, or *""*. The former two are converted to a string using the date format passed to the constructor. The latter two are converted to "".

If there's no database record, consult keyword arg *default*. It it's the string "today", use todays's date. Otherwise it can be any of the values allowed above. If no default is specified, the text field is initialized to "".

Hint: you may wish to attach a Javascript calendar to the field.

file(*name*, ***kw*)

Build a file upload field.

User agents may or may not respect the contents of the 'value' attribute.

hidden(*name*, ***kw*)

Build a hidden HTML field.

password(*name*, ***kw*)

Build a password field.

This is the same as a text box but the value will not be shown on the screen as the user types.

radio(*name*, *checked_value*, *label=None*, ***kw*)

Build a radio button.

The radio button will initially be selected if the database value equals `checked_value`. On form submission the value will be `checked_value` if the button was selected, or `""` otherwise.

In case of a ModelTags object that is created from scratch (e.g. `new_employee=ModelTags(None)`) the option that should be checked can be set by the 'default' parameter. As in: `new_employee.radio('status', checked_value=7, default=7)`

The control's 'id' attribute will be modified as follows:

1. If not specified but an 'id_format' was given to the constructor, generate an ID based on the format.
2. If an ID was passed in or was generated by step (1), append an underscore and the checked value. Before appending the checked value, lowercase it, change any spaces to `"_"`, and remove any non-alphanumeric characters except underscores and hyphens.
3. If no ID was passed or generated by step (1), the radio button will not have an 'id' attribute.

To display multiple radio buttons in a group, see webhelper.containers.distribute().

select(*name*, *options*, ***kw*)
Build a dropdown select box or list box.

See the `select()` function for the meaning of the arguments.

If the corresponding database value is not a list or tuple, it's wrapped in a one-element list. But if it's "" or `None`, an empty list is substituted. This is to accommodate multiselect lists, which may have multiple values selected.

text(*name*, ***kw*)
Build a text box.

textarea(*name*, ***kw*)
Build a rectangular text area.

9.3 Hyperlinks

webhelpers.html.tags.**link_to**(*label*, *url=''*, ***attrs*)
Create a hyperlink with the given text pointing to the URL.

If the label is `None` or empty, the URL will be used as the label.

This function does not modify the URL in any way. The label will be escaped if it contains HTML markup. To prevent escaping, wrap the label in a `webhelpers.html.literal()`.

webhelpers.html.tags.**link_to_if**(*condition*, *label*, *url=''*, ***attrs*)
Same as `link_to` but return just the label if the condition is false.

This is useful in a menu when you don't want the current option to be a link. The condition will be something like: `actual_value != value_of_this_menu_item`.

webhelpers.html.tags.**link_to_unless**(*condition*, *label*, *url=''*, ***attrs*)
The opposite of `link_to`. Return just the label if the condition is true.

9.4 Table tags

webhelpers.html.tags.**th_sortable**(*current_order*, *column_order*, *label*, *url*, *class_if_sort_column='sort'*, *class_if_not_sort_column=None*, *link_attrs=None*, *name='th'*, ***attrs*)
<th> for a "click-to-sort-by" column.

Convenience function for a sortable column. If this is the current sort column, just display the label and set the cell's class to `class_if_sort_column`.

`current_order` is the table's current sort order. `column_order` is the value pertaining to this column. In other words, if the two are equal, the table is currently sorted by this column.

If this is the sort column, display the label and set the <th>'s class to `class_if_sort_column`.

If this is not the sort column, display an <a> hyperlink based on `label`, `url`, and `link_attrs` (a dict), and set the <th>'s class to `class_if_not_sort_column`.

`url` is the literal href= value for the link. Pylons users would typically pass something like `url=h.url_for("mypage", sort="date")`.

`**attrs` are additional attributes for the <th> tag.

If you prefer a <td> tag instead of <th>, pass `name="td"`.

To change the sort order via client-side Javascript, pass `url=None` and the appropriate Javascript attributes in `link_attrs`.

Examples:

```
>>> sort = "name"
>>> th_sortable(sort, "name", "Name", "?sort=name")
literal(u'<th class="sort">Name</th>')
>>> th_sortable(sort, "date", "Date", "?sort=date")
literal(u'<th><a href="?sort=date">Date</a></th>')
>>> th_sortable(sort, "date", "Date", None, link_attrs={"onclick": "myfunc()"})
literal(u'<th><a onclick="myfunc()">Date</a></th>')
```

9.5 Other non-form tags

`webhelpers.html.tags.`**`ol`**(*items*, *default=literal(u'')*, *li_attrs=None*, ***attrs*)

Return an ordered list with each item wrapped in <li>.

`items` list of strings.

`default` value returned _instead of the _ if there are no items in the list. If `None`, return an empty <ol>.

`li_attrs` dict of attributes for the <li> tags.

Examples:

```
>>> ol(["foo", "bar"])
literal(u'<ol>\n<li>foo</li>\n<li>bar</li>\n</ol>')
>>> ol(["A", "B"], li_attrs={"class_": "myli"}, class_="mylist")
literal(u'<ol class="mylist">\n<li class="myli">A</li>\n<li class="myli">B</li>\n</ol>')
>>> ol([])
literal(u'')
```

`webhelpers.html.tags.`**`ul`**(*items*, *default=None*, *li_attrs=None*, ***attrs*)

Return an unordered list with each item wrapped in <li>.

`items` list of strings.

`default` value returned _instead of the _ if there are no items in the list. If `None`, return an empty <ul>.

`li_attrs` dict of attributes for the <li> tags.

Examples:

```
>>> ul(["foo", "bar"])
literal(u'<ul>\n<li>foo</li>\n<li>bar</li>\n</ul>')
>>> ul(["A", "B"], li_attrs={"class_": "myli"}, class_="mylist")
literal(u'<ul class="mylist">\n<li class="myli">A</li>\n<li class="myli">B</li>\n</ul>')
>>> ul([])
literal(u'<ul></ul>')
```

```
    >>> ul([], default="")
    ''
    >>> ul([], default=literal('<span class="no-data">No data</span>'))
    literal(u'<span class="no-data">No data</span>')
    >>> ul(["A"], default="NOTHING")
    literal(u'<ul>\n<li>A</li>\n</ul>')
```

webhelpers.html.tags.**image**(*url*, *alt*, *width=None*, *height=None*, *path=None*, *use_pil=False*, ***attrs*)

Return an image tag for the specified `source`.

url The URL of the image. (This must be the exact URL desired. A previous version of this helper added magic prefixes; this is no longer the case.)

alt The img's alt tag. Non-graphical browsers and screen readers will output this instead of the image. If the image is pure decoration and uninteresting to non-graphical users, pass "". To omit the alt tag completely, pass None.

width The width of the image, default is not included.

height The height of the image, default is not included.

path Calculate the width and height based on the image file at `path` if possible. May not be specified if `width` or `height` is specified. The results are also written to the debug log for troubleshooting.

use_pil If true, calcuate the image dimensions using the Python Imaging Library, which must be installed. Otherwise use a pure Python algorithm which understands fewer image formats and may be less accurate. This flag controls whether `webhelpers.media.get_dimensions_pil` or `webhelpers.media.get_dimensions` is called. It has no effect if `path` is not specified.

Examples:

```
    >>> image('/images/rss.png', 'rss syndication')
    literal(u'<img alt="rss syndication" src="/images/rss.png" />')

    >>> image('/images/xml.png', "")
    literal(u'<img alt="" src="/images/xml.png" />')

    >>> image("/images/icon.png", height=16, width=10, alt="Edit Entry")
    literal(u'<img alt="Edit Entry" height="16" src="/images/icon.png" width="10" />')

    >>> image("/icons/icon.gif", alt="Icon", width=16, height=16)
    literal(u'<img alt="Icon" height="16" src="/icons/icon.gif" width="16" />')

    >>> image("/icons/icon.gif", None, width=16)
    literal(u'<img alt="" src="/icons/icon.gif" width="16" />')
```

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

webhelpers.html.tags.**stylesheet_link**(**urls*, ***attrs*)

Return CSS link tags for the specified stylesheet URLs.

`urls` should be the exact URLs desired. A previous version of this helper added magic prefixes; this is no longer the case.

Examples:

```
>>> stylesheet_link('/stylesheets/style.css')
literal(u'<link href="/stylesheets/style.css" media="screen" rel="stylesheet" type="text/css" />

>>> stylesheet_link('/stylesheets/dir/file.css', media='all')
literal(u'<link href="/stylesheets/dir/file.css" media="all" rel="stylesheet" type="text/css" />
```

webhelpers.html.tags.**javascript_link**(**urls*, ***attrs*)

Return script include tags for the specified javascript URLs.

`urls` should be the exact URLs desired. A previous version of this helper added magic prefixes; this is no longer the case.

Specify the keyword argument `defer=True` to enable the script defer attribute.

Examples:

```
>>> print javascript_link('/javascripts/prototype.js', '/other-javascripts/util.js')
<script src="/javascripts/prototype.js" type="text/javascript"></script>
<script src="/other-javascripts/util.js" type="text/javascript"></script>

>>> print javascript_link('/app.js', '/test/test.1.js')
<script src="/app.js" type="text/javascript"></script>
<script src="/test/test.1.js" type="text/javascript"></script>
```

webhelpers.html.tags.**auto_discovery_link**(*url*, *feed_type='rss'*, ***attrs*)

Return a link tag allowing auto-detecting of RSS or ATOM feed.

The auto-detection of feed for the current page is only for browsers and news readers that support it.

url The URL of the feed. (This should be the exact URLs desired. A previous version of this helper added magic prefixes; this is no longer the case.)

feed_type The type of feed. Specifying 'rss' or 'atom' automatically translates to a type of 'application/rss+xml' or 'application/atom+xml', respectively. Otherwise the type is used as specified. Defaults to 'rss'.

Examples:

```
>>> auto_discovery_link('http://feed.com/feed.xml')
literal(u'<link href="http://feed.com/feed.xml" rel="alternate" title="RSS" type="application/rs

>>> auto_discovery_link('http://feed.com/feed.xml', feed_type='atom')
literal(u'<link href="http://feed.com/feed.xml" rel="alternate" title="ATOM" type="application/a

>>> auto_discovery_link('app.rss', feed_type='atom', title='atom feed')
literal(u'<link href="app.rss" rel="alternate" title="atom feed" type="application/atom+xml" />'

>>> auto_discovery_link('/app.html', feed_type='text/html')
literal(u'<link href="/app.html" rel="alternate" title="" type="text/html" />')
```

class webhelpers.html.tags.**Doctype**

Document type declarations for HTML and XHTML.

html4(*subtype='transitional'*, *version='4.01'*)

Create a <!DOCTYPE> for HTML 4.

Usage:

```
>>> Doctype().html4()
literal(u'<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/
>>> Doctype().html4("strict")
literal(u'<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Strict//EN" "http://www.w3.org/TR/htm
```

```
>>> Doctype().html4("frameset")
literal(u'<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN" "http://www.w3.org/TR/h
```

html5()
Create a <!DOCTYPE> for HTML 5.

Usage:

```
>>> Doctype().html5()
literal(u'<!doctype html>')
```

xhtml1(*subtype='transitional'*, *version='1.0'*)
Create a <!DOCTYPE> for XHTML 1.

Usage:

```
>>> Doctype().xhtml1()
literal(u'<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/
>>> Doctype().xhtml1("strict")
literal(u'<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xht
>>> Doctype().xhtml1("frameset")
literal(u'<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN" "http://www.w3.org/TR/x
```

webhelpers.html.tags.**xml_declaration**(*version='1.0'*, *encoding='utf-8'*)
Create an XML declaration.

Usage:

```
>>> xml_declaration()
literal(u'<?xml version="1.0" encoding="utf-8" ?>')
```

9.7 Utility functions

webhelpers.html.tags.**css_classes**(*value_condition_pairs*)
Add CSS classes to a tag programmatically.

This helper is meant to be used as the `class_` argument to a tag helper.

The argument is an iterable of `(class, condition)` pairs, where each `class` is a string and `condition` is a boolean. The function returns a space-separated list of classes whose conditions were true.

If all conditions are false, return `None`. This tells the caller to suppress the "class" attribute entirely.

Examples:

```
>>> arg = [("first", False), ("even", True)]
>>> HTML.td("My content.", class_=css_classes(arg))
literal(u'<td class="even">My content.</td>')
>>> arg = [("first", True), ("even", True)]
>>> HTML.td("My content.", class_=css_classes(arg))
literal(u'<td class="first even">My content.</td>')
>>> arg = [("first", False), ("even", False)]
>>> HTML.td("My content.", class_=css_classes(arg))
literal(u'<td>My content.</td>')
```

webhelpers.html.tags.**convert_boolean_attrs**(*attrs*, *bool_attrs*)
Convert boolean values into proper HTML attributes.

`attrs` is a dict of HTML attributes, which will be modified in place.

`bool_attrs` is a list of attribute names.

For every element in `bool_attrs`, I look for a corresponding key in `attrs`. If its value is true, I change the value to match the key. For example, I convert `selected=True` into `selected="selected"`. If the value is false, I delete the key.

CHAPTER 10

webhelpers.html.tools

HTML helpers that are more than just simple tags.

There are no helpers to prettify HTML or canonicalize whitespace because BeautifulSoup and HTMLTidy handle this well.

webhelpers.html.tools.**auto_link**(*text*, *link='all'*, ***href_attrs*)
Turn all urls and email addresses into clickable links.

link Used to determine what to link. Options are "all", "email_addresses", or "urls"

href_attrs Additional attributes for generated <a> tags.

Example:

```
>>> auto_link("Go to http://www.planetpython.com and say hello to guido@python.org")
literal(u'Go to <a href="http://www.planetpython.com">http://www.planetpython.com</a> and say he
```

webhelpers.html.tools.**button_to**(*name*, *url=''*, ***html_attrs*)
Generate a form containing a sole button that submits to `url`.

Use this method instead of `link_to` for actions that do not have the safe HTTP GET semantics implied by using a hypertext link.

The parameters are the same as for `link_to`. Any `html_attrs` that you pass will be applied to the inner `input` element. In particular, pass

disabled = True/False

as part of `html_attrs` to control whether the button is disabled. The generated form element is given the class 'button-to', to which you can attach CSS styles for display purposes.

The submit button itself will be displayed as an image if you provide both `type` and `src` as followed:

type='image', src='icon_delete.gif'

The `src` path should be the exact URL desired. A previous version of this helper added magical prefixes but this is no longer the case.

Example 1:

```
# inside of controller for "feeds"
>> button_to("Edit", url(action='edit', id=3))
<form method="post" action="/feeds/edit/3" class="button-to">
<div><input value="Edit" type="submit" /></div>
</form>
```

Example 2:

```
>> button_to("Destroy", url(action='destroy', id=3),
.. method='DELETE')
<form method="POST" action="/feeds/destroy/3"
 class="button-to">
<div>
    <input type="hidden" name="_method" value="DELETE" />
    <input value="Destroy" type="submit" />
</div>
</form>
```

Example 3:

```
# Button as an image.
>> button_to("Edit", url(action='edit', id=3), type='image',
.. src='icon_delete.gif')
<form method="POST" action="/feeds/edit/3" class="button-to">
<div><input alt="Edit" src="/images/icon_delete.gif"
 type="image" value="Edit" /></div>
</form>
```

Note: This method generates HTML code that represents a form. Forms are "block" content, which means that you should not try to insert them into your HTML where only inline content is expected. For example, you can legally insert a form inside of a `div` or `td` element or in between `p` elements, but not in the middle of a run of text, nor can you place a form within another form. (Bottom line: Always validate your HTML before going public.)

Changed in WebHelpers 1.2: Preserve case of "method" arg for XHTML compatibility. E.g., "POST" or "PUT" causes *method="POST"*; "post" or "put" causes *method="post"*.

`webhelpers.html.tools.`**`js_obfuscate`**(*content*)

Obfuscate data in a Javascript tag.

Example:

```
>>> js_obfuscate("<input type='hidden' name='check' value='valid' />")
literal(u'<script type="text/javascript">\n//<![CDATA[\neval(unescape(\'%64%6f%63%75%6d%65%6e%74
```

`webhelpers.html.tools.`**`highlight`**(*text*, *phrase*, *highlighter=None*, *case_sensitive=False*, *class_='highlight'*, ***attrs*)

Highlight all occurrences of `phrase` in `text`.

This inserts "..." around every occurrence.

Arguments:

`text`: The full text.

`phrase`: A phrase to find in the text. This may be a string, a list of strings, or a compiled regular expression. If a string, it's regex-escaped and compiled. If a list, all of the strings will be highlighted. This is done by regex-escaping all elements and then joining them using the regex "|" token.

`highlighter`: Deprecated. A replacement expression for the regex substitution. This was deprecated because it bypasses the HTML builder and creates tags via string mangling. The previous default was '1', which mimics the normal behavior of this function. `phrase` must be a string if `highlighter` is specified. Overrides `class_` and `attrs_` arguments.

`case_sensitive`: If false (default), the phrases are searched in a case-insensitive manner. No effect if `phrase` is a regex object.

`class_`: CSS class for the tag.

****attrs:** Additional HTML attributes for the tag.

Changed in WebHelpers 1.0b2: new implementation using HTML builder. Allow `phrase` to be list or regex. Deprecate `highlighter` and change its default value to None. Add `case_sensitive`, `class_`, and `**attrs` arguments.

webhelpers.html.tools.**mail_to**(*email_address*, *name=None*, *cc=None*, *bcc=None*, *subject=None*, *body=None*, *replace_at=None*, *replace_dot=None*, *encode=None*, ***html_attrs*)

Create a link tag for starting an email to the specified `email_address`.

This `email_address` is also used as the name of the link unless `name` is specified. Additional HTML options, such as class or id, can be passed in the `html_attrs` hash.

You can also make it difficult for spiders to harvest email address by obfuscating them.

Examples:

```
>>> mail_to("me@domain.com", "My email", encode = "javascript")
literal(u'<script type="text/javascript">\n//<![CDATA[\neval(unescape(\'%64%6f%63%75%6d%65%6e%74

>>> mail_to("me@domain.com", "My email", encode = "hex")
literal(u'<a href="&#109;&#97;&#105;&#108;&#116;&#111;&#58;%6d%65@%64%6f%6d%61%69%6e.%63%6f%6d">
```

You can also specify the cc address, bcc address, subject, and body parts of the message header to create a complex e-mail using the corresponding `cc`, `bcc`, `subject`, and `body` keyword arguments. Each of these options are URI escaped and then appended to the `email_address` before being output. **Be aware that javascript keywords will not be escaped and may break this feature when encoding with javascript.**

Examples:

```
>>> mail_to("me@domain.com", "My email", cc="ccaddress@domain.com", bcc="bccaddress@domain.com",
literal(u'<a href="mailto:me@domain.com?cc=ccaddress%40domain.com&amp;bcc=bccaddress%40domain.cc
```

webhelpers.html.tools.**strip_links**(*text*)

Strip link tags from `text` leaving just the link label.

Example:

```
>>> strip_links('<a href="something">else</a>')
'else'
```

webhelpers.html.tools.**strip_tags**(*text*)

Delete any HTML tags in the text, leaving their contents intact. Convert newlines to spaces, and
 to newlines.

Example::

```
>>> strip_tags('Text <em>emphasis</em> <script>Javascript</script>.')
'Text emphasis Javascript.'
>>> strip_tags('Ordinary <!-- COMMENT! --> text.')
'Ordinary  COMMENT!  text.'
>>> strip_tags('Line\n1<br />Line 2')
'Line 1\nLine 2'
```

Implementation copied from `WebOb`.

`webhelpers.html.converters` contains more sophisticated versions of this.

CHAPTER 11

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 12

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.

aliases = {}

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 13

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
```

message = ''

class webhelpers.misc.**OverwriteError**(*filename*, *message="not overwriting '%s'"*)
Refusing to overwrite an existing file or directory.

CHAPTER 14

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], sample=False)
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], sample=False)
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 15

webhelpers.paginate

15.1 paginate: a module to help split up lists or results from ORM queries

15.1.1 What is pagination?

This module helps dividing large lists of items into pages. The user is shown one page at a time and can navigate to other pages. Imagine you are offering a company phonebook and let the user search the entries. If the search result contains 23 entries but you may want to display no more than 10 entries at once. The first page contains entries 1-10, the second 11-20 and the third 21-23. See the documentation of the "Page" class for more information.

15.1.2 How do I use it?

One page of items is represented by the *Page* object. A *Page* gets initialized with at least two arguments and usually three:

- The collection of items to pick a range from.
- The page number we want to display. (Default is 1: the first page.)
- A URL generator callback. (This tells what the URLs to other pages are. It's required if using the `pager()` method, although it may be omitted under Pylons for backward compatibility. It is required for Pyramid.)

Here's an interactive example.

First we'll create a URL generator using the basic `PageURL` class, which works with all frameworks and has no dependencies. It creates URLs by overriding the 'page' query parameter.

```
# Instantiate the URL generator, and call it to see what it does.
>>> url_for_page = PageURL("/articles/2013", {"page": "3"})
>>> url_for_page(page=2)
'/articles/2013?page=2'
```

Now we can create a collection and instantiate the Page:

```
# Create a sample collection of 1000 items
>>> my_collection = range(1000)

# Create a Page object for the 3rd page (20 items per page is the default)
>>> my_page = Page(my_collection, page=3, url=url_for_page)

# The page object can be printed directly to get its details
```

```
>>> my_page
Page:
Collection type:  <type 'list'>
(Current) page:   3
First item:       41
Last item:        60
First page:       1
Last page:        50
Previous page:    2
Next page:        4
Items per page:   20
Number of items:  1000
Number of pages:  50
<BLANKLINE>

# Print a list of items on the current page
>>> my_page.items
[40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59]

# The *Page* object can be used as an iterator:
>>> for my_item in my_page: print my_item,
40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59

# The .pager() method returns an HTML fragment with links to surrounding
# pages.
# [The ">>" prompt is to hide untestable examples from doctest.]
>> my_page.pager()
1 2 [3] 4 5 .. 50       (this is actually HTML)

# The pager can be customized:
>> my_page.pager('$link_previous ~3~ $link_next (Page $page of $page_count)')
1 2 [3] 4 5 6 .. 50 > (Page 3 of 50)
```

There are many parameters that customize the Page's behavor. See the documentation on `Page` and `Page.pager()`.

15.1.3 URL generator

The constructor's `url` argument is a callback that returns URLs to other pages. It's required when using the `Page.pager()` method except under Pylons, where it will fall back to `pylons.url.current` (Pylons 1) and then `routes.url_for` (Pylons 0.9.7). If none of these are available, you'll get an exception "NotImplementedError: no URL generator available".

WebHelpers 1.3 introduces a few URL generators for convenience. **PageURL** is described above. **PageURL_WebOb** takes a `webobb.Request` object, and is suitable for Pyramid, Pylons, TurboGears, and other frameworks that have a WebOb-compatible Request object. Both of these classes assume that the page number is in the 'page' query parameter.

Here's an example for Pyramid and other WebOb-compatible frameworks:

```
# Assume ``request`` is the current request.
import webhelpers.paginate as paginate
current_page = int(request.params["page"])
q = SOME_SQLALCHEMY_QUERY
page_url = paginate.PageURL_WebOb(request)
records = paginate.Page(q, current_page, url=page_url)
```

If the page number is in the URL path, you'll have to use a framework-specific URL generator. For instance, in Pyramid if the current route is "/articles/{id}/page/{page}" and the current URL is "/articles/ABC/page/3?print=1",

you can use Pyramid's "current_route_url" function as follows:

```
# Assume ``request`` is the current request.
import webhelpers.paginate as paginate
from pyramid.url import current_route_url
def page_url(page):
    return current_route_url(request, page=page, _query=request.GET)
q = SOME_SQLALCHEMY_QUERY
current_page = int(request.matchdict["page"])
records = Page(q, current_page, url=page_url)
```

This overrides the 'page' path variable, while leaving the 'id' variable and the query string intact.

The callback API is simple.

1. It must accept an integer argument 'page', which will be passed by name.
2. It should return the URL for that page.
3. If you're using AJAX 'partial' functionality described in the `Page.pager` docstring, the callback should also accept a 'partial' argument and, if true, set a query parameter 'partial=1'.
4. If you use the 'page_param' or 'partial_param' argument to `Page.pager`, the 'page' and 'partial' arguments will be renamed to whatever you specify. In this case, the callback would also have to expect these other argument names.

The supplied classes adhere to this API in their `.__call__` method, all except the fourth condition. So you can use their instances as callbacks as long as you don't use 'page_param' or 'partial_param'.

For convenience in writing callbacks that update the 'page' query parameter, a `make_page_url` function is available that assembles the pieces into a complete URL. Other callbacks may find `webhelpers.utl.update_params` useful, which overrides query parameters on a more general basis.

15.1.4 Can I use AJAX / AJAH?

Yes. See *partial_param* and *onclick* in `Page.pager()`.

15.1.5 Notes

Page numbers and item numbers start at 1. This concept has been used because users expect that the first page has number 1 and the first item on a page also has number 1. So if you want to use the page's items by their index number please note that you have to subtract 1.

This module is the successor to the obsolete `webhelpers.pagination` module. It is **NOT** API compatible.

This module is based on the code from http://workaround.org/cgi-bin/hg-paginate that is known at the "Paginate" module on PyPI. It was written by Christoph Haas <email@christoph-haas.de>, and modified by Christoph Haas and Mike Orr for WebHelpers. (c) 2007-2011.

15.2 Page Objects

class `webhelpers.paginate.`**`Page`**(*collection*, *page=1*, *items_per_page=20*, *item_count=None*, *sqlalchemy_session=None*, *presliced_list=False*, *url=None*, ***kwargs*)

A list/iterator of items representing one page in a larger collection.

An instance of the "Page" class is created from a collection of things. The instance works as an iterator running from the first item to the last item on the given page. The collection can be:

- a sequence
- an SQLAlchemy query - e.g.: Session.query(MyModel)
- an SQLAlchemy select - e.g.: sqlalchemy.select([my_table])

A "Page" instance maintains pagination logic associated with each page, where it begins, what the first/last item on the page is, etc. The pager() method creates a link list allowing the user to go to other pages.

WARNING: Unless you pass in an item_count, a count will be performed on the collection every time a Page instance is created. If using an ORM, it's advised to pass in the number of items in the collection if that number is known.

Instance attributes:

original_collection Points to the collection object being paged through

item_count Number of items in the collection

page Number of the current page

items_per_page Maximal number of items displayed on a page

first_page Number of the first page - starts with 1

last_page Number of the last page

page_count Number of pages

items Sequence/iterator of items on the current page

first_item Index of first item on the current page - starts with 1

last_item Index of last item on the current page

pager (*format='~2~'*, *page_param='page'*, *partial_param='partial'*, *show_if_single_page=False*, *separator=' '*, *onclick=None*, *symbol_first='<<'*, *symbol_last='>>'*, *symbol_previous='<'*, *symbol_next='>'*, *link_attr={'class': 'pager_link'}*, *curpage_attr={'class': 'pager_curpage'}*, *dotdot_attr={'class': 'pager_dotdot'}*, ***kwargs*)

Return string with links to other pages (e.g. "1 2 [3] 4 5 6 7").

format: Format string that defines how the pager is rendered. The string can contain the following $-tokens that are substituted by the string.Template module:

- $first_page: number of first reachable page
- $last_page: number of last reachable page
- $page: number of currently selected page
- $page_count: number of reachable pages
- $items_per_page: maximal number of items per page
- $first_item: index of first item on the current page
- $last_item: index of last item on the current page
- $item_count: total number of items
- $link_first: link to first page (unless this is first page)
- $link_last: link to last page (unless this is last page)
- $link_previous: link to previous page (unless this is first page)

- $link_next: link to next page (unless this is last page)

To render a range of pages the token '~3~' can be used. The number sets the radius of pages around the current page. Example for a range with radius 3:

'1 .. 5 6 7 [8] 9 10 11 .. 500'

Default: '~2~'

symbol_first String to be displayed as the text for the %(link_first)s link above.

Default: '<<'

symbol_last String to be displayed as the text for the %(link_last)s link above.

Default: '>>'

symbol_previous String to be displayed as the text for the %(link_previous)s link above.

Default: '<'

symbol_next String to be displayed as the text for the %(link_next)s link above.

Default: '>'

separator: String that is used to separate page links/numbers in the above range of pages.

Default: ' '

page_param: The name of the parameter that will carry the number of the page the user just clicked on. The parameter will be passed to a url_for() call so if you stay with the default ':controller/:action/:id' routing and set page_param='id' then the :id part of the URL will be changed. If you set page_param='page' then url_for() will make it an extra parameters like ':controller/:action/:id?page=1'. You need the page_param in your action to determine the page number the user wants to see. If you do not specify anything else the default will be a parameter called 'page'.

Note: If you set this argument and are using a URL generator callback, the callback must accept this name as an argument instead of 'page'. callback, becaust the callback requires its argument to be 'page'. Instead the callback itself can return any URL necessary.

partial_param: When using AJAX/AJAH to do partial updates of the page area the application has to know whether a partial update (only the area to be replaced) or a full update (reloading the whole page) is required. So this parameter is the name of the URL parameter that gets set to 1 if the 'onclick' parameter is used. So if the user requests a new page through a Javascript action (onclick) then this parameter gets set and the application is supposed to return a partial content. And without Javascript this parameter is not set. The application thus has to check for the existence of this parameter to determine whether only a partial or a full page needs to be returned. See also the examples in this modules docstring.

Default: 'partial'

Note: If you set this argument and are using a URL generator callback, the callback must accept this name as an argument instead of 'partial'.

show_if_single_page: if True the navigator will be shown even if there is only one page

Default: False

link_attr (optional) A dictionary of attributes that get added to A-HREF links pointing to other pages. Can be used to define a CSS style or class to customize the look of links.

Example: { 'style':'border: 1px solid green' }

Default: { 'class':'pager_link' }

curpage_attr (optional) A dictionary of attributes that get added to the current page number in the pager (which is obviously not a link). If this dictionary is not empty then the elements will be wrapped in a SPAN tag with the given attributes.

Example: { 'style':'border: 3px solid blue' }

Default: { 'class':'pager_curpage' }

dotdot_attr (optional) A dictionary of attributes that get added to the '..' string in the pager (which is obviously not a link). If this dictionary is not empty then the elements will be wrapped in a SPAN tag with the given attributes.

Example: { 'style':'color: #808080' }

Default: { 'class':'pager_dotdot' }

onclick (optional) This paramter is a string containing optional Javascript code that will be used as the 'onclick' action of each pager link. It can be used to enhance your pager with AJAX actions loading another page into a DOM object.

In this string the variable '$partial_url' will be replaced by the URL linking to the desired page with an added 'partial=1' parameter (or whatever you set 'partial_param' to). In addition the '$page' variable gets replaced by the respective page number.

Note that the URL to the destination page contains a 'partial_param' parameter so that you can distinguish between AJAX requests (just refreshing the paginated area of your page) and full requests (loading the whole new page).

[Backward compatibility: you can use '%s' instead of '$partial_url']

jQuery example: "$('#my-page-area').load('$partial_url'); return false;"

Yahoo UI example:

"YAHOO.util.Connect.asyncRequest('GET','$partial_url',{
success:function(o){YAHOO.util.Dom.get('#my-page-area').innerHTML=o.responseText;}
},null); return false;"

scriptaculous example:

"new Ajax.Updater('#my-page-area', '$partial_url', {asynchronous:true, evalScripts:true}); return false;"

ExtJS example: "Ext.get('#my-page-area').load({url:'$partial_url'}); return false;"

Custom example: "my_load_page($page)"

Additional keyword arguments are used as arguments in the links. Otherwise the link will be created with url_for() which points to the page you are currently displaying.

15.3 URL generators

class webhelpers.paginate.**PageURL**(*path*, *params*)
A simple page URL generator for any framework.

__call__(*page*, *partial=False*)
Generate a URL for the specified page.

class webhelpers.paginate.**PageURL_WebOb**(*request*, *qualified=False*)
A page URL generator for WebOb-compatible Request objects.

I derive new URLs based on the current URL but overriding the 'page' query parameter.

I'm suitable for Pyramid, Pylons, and TurboGears, as well as any other framework whose Request object has 'application_url', 'path', and 'GET' attributes that behave the same way as `webob.Request`'s.

__call__(*page*, *partial=False*)
Generate a URL for the specified page.

`webhelpers.paginate.`**make_page_url**(*path*, *params*, *page*, *partial=False*, *sort=True*)
A helper function for URL generators.

I assemble a URL from its parts. I assume that a link to a certain page is done by overriding the 'page' query parameter.

`path` is the current URL path, with or without a "scheme://host" prefix.

`params` is the current query parameters as a dict or dict-like object.

`page` is the target page number.

If `partial` is true, set query param 'partial=1'. This is to for AJAX calls requesting a partial page.

If `sort` is true (default), the parameters will be sorted. Otherwise they'll be in whatever order the dict iterates them.

CHAPTER 16

webhelpers.text

Functions that output text (not HTML).

Helpers for filtering, formatting, and transforming strings.

webhelpers.text.**chop_at**(*s*, *sub*, *inclusive=False*)

Truncate string `s` at the first occurrence of `sub`.

If `inclusive` is true, truncate just after `sub` rather than at it.

```
>>> chop_at("plutocratic brats", "rat")
'plutoc'
>>> chop_at("plutocratic brats", "rat", True)
'plutocrat'
```

webhelpers.text.**collapse**(*string*, *character=' '*)

Removes specified character from the beginning and/or end of the string and then condenses runs of the character within the string.

Based on Ruby's stringex package (http://github.com/rsl/stringex/tree/master)

webhelpers.text.**convert_accented_entities**(*string*)

Converts HTML entities into the respective non-accented letters.

Examples:

```
>>> convert_accented_entities("&aacute;")
'a'
>>> convert_accented_entities("&ccedil;")
'c'
>>> convert_accented_entities("&egrave;")
'e'
>>> convert_accented_entities("&icirc;")
'i'
>>> convert_accented_entities("&oslash;")
'o'
>>> convert_accented_entities("&uuml;")
'u'
```

Note: This does not do any conversion of Unicode/ASCII accented-characters. For that functionality please use unidecode.

Based on Ruby's stringex package (http://github.com/rsl/stringex/tree/master)

webhelpers.text.**convert_misc_entities**(*string*)

Converts HTML entities (taken from common Textile formattings) into plain text formats

Note: This isn't an attempt at complete conversion of HTML entities, just those most likely to be generated by Textile.

Based on Ruby's stringex package (http://github.com/rsl/stringex/tree/master)

webhelpers.text.**excerpt**(*text*, *phrase*, *radius=100*, *excerpt_string='...'*)
Extract an excerpt from the `text`, or '' if the phrase isn't found.

phrase Phrase to excerpt from `text`

radius How many surrounding characters to include

excerpt_string Characters surrounding entire excerpt

Example:

```
>>> excerpt("hello my world", "my", 3)
'...lo my wo...'
```

webhelpers.text.**lchop**(*s*, *sub*)
Chop `sub` off the front of `s` if present.

```
>>> lchop("##This is a comment.##", "##")
'This is a comment.##'
```

The difference between `lchop` and `s.lstrip` is that `lchop` strips only the exact prefix, while `s.lstrip` treats the argument as a set of leading characters to delete regardless of order.

webhelpers.text.**plural**(*n*, *singular*, *plural*, *with_number=True*)
Return the singular or plural form of a word, according to the number.

If `with_number` is true (default), the return value will be the number followed by the word. Otherwise the word alone will be returned.

Usage:

```
>>> plural(2, "ox", "oxen")
'2 oxen'
>>> plural(2, "ox", "oxen", False)
'oxen'
```

webhelpers.text.**rchop**(*s*, *sub*)
Chop `sub` off the end of `s` if present.

```
>>> rchop("##This is a comment.##", "##")
'##This is a comment.'
```

The difference between `rchop` and `s.rstrip` is that `rchop` strips only the exact suffix, while `s.rstrip` treats the argument as a set of trailing characters to delete regardless of order.

webhelpers.text.**remove_formatting**(*string*)
Simplify HTML text by removing tags and several kinds of formatting.

If the `unidecode` package is installed, it will also transliterate non-ASCII Unicode characters to their nearest pronunciation equivalent in ASCII.

Based on Ruby's stringex package (http://github.com/rsl/stringex/tree/master)

webhelpers.text.**replace_whitespace**(*string*, *replace=' '*)
Replace runs of whitespace in string

Defaults to a single space but any replacement string may be specified as an argument. Examples:

```
>>> replace_whitespace("Foo       bar")
'Foo bar'
>>> replace_whitespace("Foo       bar", "-")
'Foo-bar'
```

Based on Ruby's stringex package (http://github.com/rsl/stringex/tree/master)

webhelpers.text.**series**(*items*, *conjunction='and'*, *strict_commas=True*)
Join strings using commas and a conjunction such as "and" or "or".

Examples:

```
>>> series(["A", "B", "C"])
'A, B, and C'
>>> series(["A", "B", "C"], "or")
'A, B, or C'
>>> series(["A", "B", "C"], strict_commas=False)
'A, B and C'
>>> series(["A", "B"])
'A and B'
>>> series(["A"])
'A'
>>> series([])
''
```

webhelpers.text.**strip_leading_whitespace**(*s*)
Strip the leading whitespace in all lines in `s`.

This deletes *all* leading whitespace. `textwrap.dedent` deletes only the whitespace common to all lines.

webhelpers.text.**truncate**(*text*, *length=30*, *indicator='...'*, *whole_word=False*)
Truncate `text` with replacement characters.

length The maximum length of `text` before replacement

indicator If `text` exceeds the `length`, this string will replace the end of the string

whole_word If true, shorten the string further to avoid breaking a word in the middle. A word is defined as any string not containing whitespace. If the entire text before the break is a single word, it will have to be broken.

Example:

```
>>> truncate('Once upon a time in a world far far away', 14)
'Once upon a...'
```

webhelpers.text.**urlify**(*string*)
Create a URI-friendly representation of the string

Can be called manually in order to generate an URI-friendly version of any string.

If the `unidecode` package is installed, it will also transliterate non-ASCII Unicode characters to their nearest pronounciation equivalent in ASCII.

Examples::

```
>>> urlify("Mighty Mighty Bosstones")
'mighty-mighty-bosstones'
```

Based on Ruby's stringex package (http://github.com/rsl/stringex/tree/master)

Changed in WebHelpers 1.2: urlecode the result in case it contains special characters like "?".

webhelpers.text.**wrap_paragraphs**(*text*, *width=72*)

Wrap all paragraphs in a text string to the specified width.

`width` may be an int or a `textwrap.TextWrapper` instance. The latter allows you to set other options besides the width, and is more efficient when wrapping many texts.

CHAPTER 17

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", _debug=True)
('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 18

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

Helpers for the Pylons web framework

These helpers depend on Pylons' `request`, `response`, `session` objects or some other aspect of Pylons. Most of them can be easily ported to another framework by changing the API calls.

18.2 webhelpers.pylonslib.flash

Accumulate messages to show on the next page request.

The `Flash` class is useful when you want to redirect to another page and also show a status message on that page, such as "Changes saved" or "No previous search found; returning to home page".

THE IMPLEMENTATION DEPENDS ON PYLONS. However, it can easily be adapted for another web framework.

PYRAMID USERS: use the flash methods built into Pyramid's `Session` object. This implementation is incompatible with Pyramid.

A typical Pylons application instantiates a `Flash` object in myapp/lib/helpers.py:

```
from webhelpers.pylonslib.flash import Flash as _Flash
flash = _Flash()
```

The helpers module is then imported into your controllers and templates as `h`. Whenever you want to set a message, call the instance:

```
h.flash("Record deleted.")
```

You can set additional messages too:

```
h.flash("Hope you didn't need it.")
```

Now make a place in your site template for the messages. In Mako you might do:

```
<% messages = h.flash.pop_messages() %>
% if messages:
<ul id="flash-messages">
    % for message in messages:
    <li>${message}</li>
    % endfor
</ul>
% endif
```

You can style this to look however you want:

```
ul#flash-messages {
    color: red;
    background-color: #FFFFCC;
    font-size: larger;
    font-style: italic;
    margin-left: 40px;
    padding: 4px;
    list-style: none;
    }
```

18.2.1 Multiple flash objects

You can define multiple flash objects in your application to display different kinds of messages at different places on the page. For instance, you might use the main flash object for general messages, and a second flash object for "Added dookickey" / "Removed doohickey" messages next to a doohickey manager.

18.2.2 Message categories

WebHelpers 1.0 adds message categories, contributed by Wichert Akkerman. These work like severity levels in Python's logging system. The standard categories are "*warning*", "*notice*", "*error*", and "*success*", with the default being "*notice*". The category is available in the message's `.category` attribute, and is normally used to set the container's CSS class.

This is the *only* thing it does. Calling `.pop_messages()` pops all messages in the order registered, regardless of category. It is *not* possible to pop only a certain category, or all levels above a certain level, or to group messages by category. If you want to group different kinds of messages together, or pop only certain categories while leaving other categories, you should use multiple `Flash` objects.

You can change the standard categories by overriding the `.categories` and `.default_category` class attributes, or by providing alternate values using constructor keywords.

Category example

Let's show a standard way of using flash messages in your site: we will demonstrate *self-healing messages* (similar to what Growl does on OSX) to show messages in a site.

To send a message from python just call the flash helper method:

```
h.flash(u"Settings have been saved")
```

This will tell the system to show a message in the rendered page. If you need more control you can specify a message category as well: one of *warning*, *notice*, *error* or *success*. The default category is *notice*. For example:

```
h.flash(u"Failed to send confirmation email", "warning")
```

We will use a very simple markup style: messages will be placed in a `div` with id `selfHealingFeedback` at the end of the document body. The messages are standard paragraphs with a class indicating the message category. For example:

```
<html>
  <body>
    <div id="content">
      ...
      ...
    </div>
    <div id="selfHealingFeedback">
      <p class="success">Succesfully updated your settings</p>
      <p class="warning">Failed to send confirmation email</p>
    </div>
  </body>
</html>
```

This can easily created from a template. If you are using Genshi this should work:

The needed CSS is very simple:

Choosing different colours for the categories is left as an exercise for the reader.

Next we create the javascript that will manage the needed behaviour (this implementation is based on jQuery):

```
function _SetupMessage(el) {
    var remover = function () {
        msg.animate({opacity: 0}, "slow")
           .slideUp("slow", function() { msg.remove() }); };

    msg.data("healtimer", setTimeout(remover, 10000))
       .click(function() { clearTimeout(msg.data("healtimer")); remover(); });
}

function ShowMessage(message, category) {
    if (!category)
        category="notice";

    var container = $("#selfHealingFeedback");

    if (!container.length)
        container=$("<div id='selfHealingFeedback'/>").appendTo("body");

    var msg = $("<p/>").addClass(category).html(message);
    SetupMessage(msg);
    msg.appendTo(container);
}

$(document).ready(function() {
    $("#selfHealingFeedback p").each(function() { SetupMessage($(this)); });
}
```

The `SetupMessage` function configures the desired behaviour: a message disappears after 10 seconds, or if you click on it. Removal is done using a simple animation to avoid messages jumping around on the screen.

This function is called for all messages as soon as the document has fully loaded. The `ShowMessage` function works exactly like the `flash` method in python: you can call it with a message and optionally a category and it will pop up a new message.

JSON integration

It is not unusual to perform a remote task using a JSON call and show a result message to the user. This can easily be done using a simple wrapper around the ShowMessage method:

```
function ShowJSONResponse(info) {
    if (!info.message)
        return;

    ShowMessage(info.message, info.message_category);
}
```

You can use this direct as the success callback for the jQuery AJAX method:

```
$.ajax({type: "POST",
        url:  "http://your.domain/call/json",
        dataType: "json",
        success: ShowJSONResponse
});
```

if you need to perform extra work in your callback method you can call it yourself as well, for example:

```
<form action="http://your.domain/call/form">
  <input type="hidden" name="json_url" value="http://your.domain/call/json">
  <button>Submit</button>
</form>

<sript type="text/javascript">
   $(document).ready(function() {
       $("button").click(function() {
           var button = $(this);

           button.addClass("processing");
           $.ajax({type: "POST",
                   url:  this.form["json_url"].value,
                   dataType: "json",
                   success: function(data, status) {
                       button.removeClass("processing");
                       ShowJSONResponse(data);
                    },
                    error: function(request, status, error) {
                       button.removeClass("processing");
                       ShowMessage("JSON call failed", "error");
                    }
           });

           return false;
       });
   });
</script>
```

This sets up a simple form which can be submitted normally by non-javascript enabled browsers. If a user does have javascript an AJAX call will be made to the server and the result will be shown in a message. While the call is active the button will be marked with a *processing* class.

The server can return a message by including a `message` field in its response. Optionally a `message_category` field can also be included which will be used to determine the message category. For example:

```
@jsonify
def handler(self):
   ..
   ..
   return dict(message=u"Settings successfully updated")
```

18.2.3 Classes

class webhelpers.pylonslib.flash.**Flash**(*session_key='flash'*, *categories=None*, *default_category=None*)

Accumulate a list of messages to show at the next page request.

__call__(*message*, *category=None*, *ignore_duplicate=False*)

Add a message to the session.

`message` is the message text.

`category` is the message's category. If not specified, the default category will be used. Raise `ValueError` if the category is not in the list of allowed categories.

If `ignore_duplicate` is true, don't add the message if another message with identical text has already been added. If the new message has a different category than the original message, change the original message to the new category.

pop_messages()

Return all accumulated messages and delete them from the session.

The return value is a list of `Message` objects.

class webhelpers.pylonslib.flash.**Message**(*category*, *message*)

A message returned by `Flash.pop_messages()`.

Converting the message to a string returns the message text. Instances also have the following attributes:

- `message`: the message text.
- `category`: the category specified when the message was created.

18.3 webhelpers.pylonslib.grid

This module is DEPRECATED. Please use `webhelpers.html.grid` in new applications. Support for paged grids has been added to that module in a framework-neutral way.

PYRAMID USERS: This implementation is incompatible with Pyramid. Use `webhelpers.html.grid` instead.

class webhelpers.pylonslib.grid.**PylonsGrid**(*request*, **args*, ***kw*)

Subclass of Grid that can handle header link generation for quick building of tables that support ordering of their contents, paginated results etc.

generate_header_link(*column_number*, *column*, *label_text*)

This handles generation of link and then decides to call self.default_header_ordered_column_format or self.default_header_column_format based on if current column is the one that is used for sorting or not

class webhelpers.pylonslib.grid.**PylonsObjectGrid**(*request*, **args*, ***kw*)

This grid will work well with sqlalchemy row instances

18.4 webhelpers.pylonslib.minify

Minification helpers.

This module provides enhanced versions of the `javascript_link` and `stylesheet_link` helpers in `webhelpers.html.tags`. These versions add three additional arguments:

- **minified**: If true, reduce the file size by squeezing out whitespace and other characters insignificant to the Javascript or CSS syntax.
- **combined**: If true, concatenate the specified files into one file to reduce page load time.
- **beaker_kwargs** (dict): arguments to pass to `beaker_cache`.

Dependencies: `Pylons`, `Beaker`, `jsmin`, and `cssutils` (all available in PyPI). If "jsmin" is not installed, the helper issues a warning and passes Javascript through unchanged. (Changed in WebHelpers 1.1: removed built-in "_jsmin" package due to licensing issues; details in webhelpers/pylonslib/_jsmin.py .)

PYRAMID USERS: this implementation is incompatible with Pyramid. No Pyramid-compatible implementation is currently known.

Contributed by Pedro Algarvio and Domen Kozar <ufs@ufsoft.org>. URL: http://docs.fubar.si/minwebhelpers/

`webhelpers.pylonslib.minify.`**`javascript_link`**`(`*sources*, ***options*`)`

`webhelpers.pylonslib.minify.`**`stylesheet_link`**`(`*sources*, ***options*`)`

18.5 webhelpers.pylonslib.secure_form

Secure Form Tag Helpers – For prevention of Cross-site request forgery (CSRF) attacks.

Generates form tags that include client-specific authorization tokens to be verified by the destined web app.

PYRAMID USERS: Use the csrf_token methods built into Pyramid's `Session` object. This implementation is incompatible with Pyramid.

Authorization tokens are stored in the client's session. The web app can then verify the request's submitted authorization token with the value in the client's session.

This ensures the request came from the originating page. See http://en.wikipedia.org/wiki/Cross-site_request_forgery for more information.

Pylons provides an `authenticate_form` decorator that does this verification on the behalf of controllers.

These helpers depend on Pylons' `session` object. Most of them can be easily ported to another framework by changing the API calls.

The helpers are implemented in such a way that it should be easy to create your own helpers if you are using helpers for AJAX calls.

authentication_token() returns the current authentication token, creating one and storing it in the session if it doesn't already exist.

auth_token_hidden_field() creates a hidden field (wrapped in an invisible div; I don't know if this is necessary, but the old WebHelpers had it like this) containing the authentication token.

secure_form() is form() plus auth_token_hidden_field().

`webhelpers.pylonslib.secure_form.`**`authentication_token`**`()`
Return the current authentication token, creating one if one doesn't already exist.

`webhelpers.pylonslib.secure_form.`**`auth_token_hidden_field`**`()`

webhelpers.pylonslib.secure_form.**secure_form**(*url*, *method='POST'*, *multipart=False*, ***attrs*)

Start a form tag that points the action to an url. This form tag will also include the hidden field containing the auth token.

The url options should be given either as a string, or as a `url()` function. The method for the form defaults to POST.

Options:

multipart If set to True, the enctype is set to "multipart/form-data".

method The method to use when submitting the form, usually either "GET" or "POST". If "PUT", "DELETE", or another verb is used, a hidden input with name _method is added to simulate the verb over POST.

CHAPTER 19

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.

W

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X