FormEncode Documentation

Release 0.0.0

Ian Bicking

April 21, 2023

Contents

CHAPTER 1

Introduction

FormEncode is a validation and form generation package. The validation can be used separately from the form generation. The validation works on compound data structures, with all parts being nestable. It is separate from HTTP or any other input mechanism.

Ian Bicking is the author of FormEncode.

CHAPTER 2

What's New

2.1 What's New In FormEncode 2.0

This article explains the latest changes in *FormEncode* version 2.0.0 compared to its predecessor, *FormEncode* 1.3.0

2.1.1 2.0.1

- Add support for 3.10
- use Pytest instead of Nose and Github Actions instead of Travis for tests
- Documentation updates
- Note this will be the last version to support Python 2.7. The next version will be 2.1 to signal this change. If you want to keep support for Python 2.7 update your dependencies spec to be below 2.1

2.1.2 2.0.0

- *FormEncode* can now run on Python 3.6 and higher without needing to run 2to3 first.
- *FormEncode* 2.0 is no longer compatible with Python 2.6 and 3.2 to 3.5. If you need Python 2.6 or 3.2 to 3.5 compatibility please use *FormEncode* 1.3. You might also try *FormEncode* 2.0.0a1 which supports Python 2.6 and Python 3.3-3.5.
- This will be the last major version to support Python 2.7
- Add strict flag to USPostalCode to raise error on postal codes that has too many digits instead of just truncating
- Various Python 3 fixes
- Serbian latin translation
- Changed License to MIT
- Dutch, UK, Greek and South Korean postal code format fixes

- Add postal code formats for Switzerland, Cyprus, Faroe Islands, San Marino, Ukraine and Vatican City.
- Add ISODateTimeConverter validator
- Add ability to target htmlfill to particular form or ignore a form
- Fix format errors in some translations
- The version of the library can be checked using formencode.__version__

2.2 What's New In FormEncode 1.3

This article explains the latest changes in *FormEncode* version 1.3 compared to its predecessor, *FormEncode* 1.2.5.

2.2.1 Feature Additions

- Support validation of email addresses with unicode domain names.

2.2.2 Backwards Incompatibilities

- *FormEncode* 1.3 is no longer compatible with Python 2.3, 2.4, or 2.5.

  The reason? We could not easily "straddle" Python 2 and 3 versions and support Python 2 versions older than Python 2.6. You will need Python 2.6 or better to run this version of FormEncode. If you need to use Python 2.5, you should use the most recent 1.2.X release of FormEncode.

- *FormEncode* now also runs under Python 3.2 and 3.3.

  > Note that under Python 3, the *String* validator is now identical to the *UnicodeString* validator. If you really want to convert to byte strings, use the *ByteString* validator instead.

- Validation of email addresses using *resolve_domain* option now requires the dnspython third party library instead of pyDNS. pyDNS also does not support Python 3.
- The *FancyValidator* methods *_to_python*, *_from_python*, *validate_python* and *validate_other* have been renamed to *_convert_to_python*, *_convert_from_python*, *_validate_python* and *_validate_other*, respectively. This has been done to clarify that while these methods are meant to be overridden by custom validators, they are not part of the external API. They are only helper methods that are called internally by the external methods *to_python* and *from_python*, which constitute the external API. Particularly, do not assume that *_validate_python* catches all validation errors that a call of *to_python* will catch. Please have a look at the *FancyValidator* docstring and source if you're unsure how these methods work together. For the same reason, the *CompoundValidator* method *attempt_convert* has been renamed to *_attempt_convert*. For now, the old method names will still work, but they will output deprecation warnings.
- (1.3.1) Don't use universal wheel

2.3 What's New In FormEncode 1.2.5

This article explains the latest changes in *FormEncode* version 1.2.5 as compared to its predecessor, *FormEncode* 1.2.4.

2.3.1 Project Changes

- New official BitBucket code repository at formencode/official-formencode.

2.3.2 Feature Additions

- The method *field_is_empty* was added to `formencode.validators.FormValidator` so subclasses can use the same logic for emptiness and users can override it if necessary.

2.3.3 Backwards Incompatibilities

- The view attribute is no longer considered special when scanning Compound validators and Schemas for validators.
- The `formencode.validators.RequireIfMissing` and *RequireIfPresent* form validators now use the same empty/missing logic as the *is_empty* method of `formencode.api.FancyValidator`.
- Validators can say if they accept containers (list, tuple, set, etc) and schema will actively refuse those values if a validator does not allow them.

2.3.4 Documentation Enhancements

- Superceded news with whatsnew documents that will be archived for each version. Archived all news prior to 1.2.5 in *What's New In FormEncode 0 to 1.2.4*.

2.4 What's New In FormEncode 0 to 1.2.4

Contents

2.4.1 1.2.4

- Fix packaging issue with i18n files (from Juliusz Gonera)

2.4.2 1.2.3

- Code repository moved to BitBucket at ianb/formencode.
- Allowed *formencode.validators.UnicodeString* to use different encoding of input and output, or no encoding/decoding at all.
- Fixes #2666139: DateValidator bug happening only in March under Windows in Germany :)
- Don't let *formencode.compound.Any* shortcut validation when it gets an empty value (this same change had already been made to *formencode.compound.All*).
- Really updated German translation
- Fixes #2799713: force_defaults=False and <select> fields, regarding *formencode.htmlfill.render()*. Thanks to w31rd0 for the bug report and patch to fix it.

2.4.3 1.2.2

- Added keyword argument `force_defaults` to *formencode.htmlfill.render()*; when this is True (the default) this will uncheck checkboxes, unselect select boxes, etc., when a value is missing from the default dictionary.
- Updated German translation

2.4.4 1.2.1

- Be more careful about `unicode(Invalid(...))`, to make sure it always returns unicode.
- Fix broken *formencode.national* zip code validators.
- In *formencode.national* only warn about the pycountry or TG requirement when creating validators that require them.

- Fix another *formencode.htmlfill* error due to a field with no explicit value.

2.4.5 1.2

- Added *formencode.validators.IPAddress*, validating IP addresses, from Leandro Lucarella.
- Added `formencode.api.Invalid.__unicode__()`
- In *formencode.htmlfill* use a default encoding of utf8 when handling mixed `str`/`unicode` content. Also do not modify `<input type="image">` tags (previously `src` would be overwritten, for no good reason).
- In *formencode.validators.Email* allow single-character domain names (like x.com).
- Make *formencode.validators.FieldsMatch* give a normal `Invalid` exception if you pass it a non-dictionary. Also treat all missing keys as the empty string (previously the first key was required and would raise KeyError).
- *formencode.validators.Number* works with `inf` float values (before it would raise a OverflowError).
- The `tw` locale has been renamed to the more standard `zh_TW`.
- Added Japanese and Turkish translations.
- Fixed some outdated translations and errors in Spanish and Greek translations. Translations now managed with Babel.

2.4.6 1.1

- Fixed the `is_empty()` method in *formencode.validators.FieldStorageUploadConverter*; previously it returned the opposite of the intended result.
- Added a parameter to `htmlfill.render()`: `prefix_error`. If this parameter is true (the default) then errors automatically go before the input field; if false then they go after the input field.
- Remove deprecated modules: `fields`, `formgen`, `htmlform`, `sqlformgen`, and `sqlschema`.
- Added `formencode.htmlrename`, which renames HTML inputs.
- In `formencode.htmlfill`, non-string values are compared usefully (e.g., a select box with integer values).
- The validators `Int` and `Number` both take min/max arguments (from Shannon Behrens).
- Validators based on `formencode.validators.FormValidator` will not treat `{}` as an empty (unvalidated) value.
- Some adjustments to the URL validator.
- *formencode.compound.All* does not handle empty values, instead relying on sub-validators to check for emptiness.
- Fixed the `if_missing` attribute in *formencode.foreach.ForEach*; previously it would be the same list instance, so if you modified it then it would effect future `if_missing` values (reported by Felix Schwarz).
- Added formatter to *formencode.htmlfill*, so you can use `<form:error name="field_name" formatter="ignore" />` – this will cause the error to be swallowed, not shown to the user.
- Added `formencode.validators.XRI` for validation i-names, i-numbers, URLs, etc (as used in OpenID).
- Look in `/usr/share/locale` for locale files, in addition to the normal locations.
- Quiet Python 2.6 deprecation warnings.

- Fix *formencode.validators.URL*, which was accepting illegal characters (like newlines) and did not accept http://domain:PORT/

2.4.7 1.0.1

- `chained_validators` were removed from Schema somehow; now replaced and working.
- Put in missing `htmlfill.render(error_class=...)` parameter (was documented and implemented, but `render()` did not pass it through).

2.4.8 1.0

- Added `formencode.schema.SimpleFormValidator`, which wraps a simple function to make it a validator.
- Changed the use of `chained_validators` in Schemas, so that all chained validators get run even when there are previous errors (to detect all the errors).
- While something like `Int.to_python()` worked, other methods like `Int.message(...)` didn't work. Now it does.
- Added Italian, Finnish, and Norwegian translations.

2.4.9 0.9

Backward incompatible changes

- The notion of "empty" has changed to include empty lists, dictionaries, and tuples. If you get one of these values passed into (or generated by) a validator with `not_empty=True` you can get exceptions where you didn't previously.

Enhancements

- Added support for Paste's MultiDict dictionary as input to Schema.to_python, by converting it to a normal dict via MultiDict.mixed. Previously MultiDicts wouldn't work with CompoundValidators (like ForEach)
- Added encoding parameter to htmlfill, which will handle cases when mixed str and unicode objects are used (turning all str objects into unicode)
- Include `formencode.validators.InternationalPhoneNumber` from W-Mark Kubacki.
- `validators.Int` takes `min` and `max` options (from Felix Schwarz).
- You can control the missing message (which by default is just "Missing Value") using the message `"missing"` in a validator (also from James Gardner).
- Added `validators.CADR` (for IP addresses with an optional range) and `validators.MACAddress` (from Christoph Haas).

Bug Fixes

- Be friendlier when loaded from a zip file (as with py2exe); previously only egg zip files would work.
- Fixed bug in htmlfill when a document ends with no trailing text after the last tag.

- Fix problem with HTMLParser's default unescaping routing, which only understood a very limited number of entities in attribute values.
- Fix problem with looking up A records for email addresses.
- `validators.String` now always returns strings. It also converts lists to comma-separated strings (no `[...]`), and can encode unicode if an `encoding` parameter is given. Empty values are handled better.
- `validators.UnicodeString` properly handles non-Unicode inputs.
- Make `validators.DateConverter` serialize dates properly (from James Gardner).
- Minor fix to setup.py to make FormEncode more friendly with zc.buildout.

2.4.10 0.7.1

- Set `if_missing=()` on `validators.Set`, as a missing value usually means empty for this value.
- Fix for `Email` validator that searches A records in addition to MX records (from Jacob Smullyan).
- Fixes for the `es` locale.

2.4.11 0.7

- **Backward compatibility issue**: Due to the addition of i18n (internationalization) to FormEncode, Invalid exceptions now have unicode messages. You may encounter unicode-related errors if you are mixing these messages with non-ASCII `str` strings.
- gettext-enabled branch merged in
- Fixes #1457145: Fails on URLs with port numbers
- Fixes #1559918 Schema fails to accept unicode errors
- `from formencode.validators import *` will import the `Invalid` exception now.
- `Invalid().unpack_errors(encode_variables=True)` now filters out None values (which `ForEach` can produce even for keys with no errors).

2.4.12 0.6

- `String(min=1)` implies `not_empty` (which seems more intuitive)
- Added `list_char` and `dict_char` arguments to `Invalid.unpack_errors` (passed through to `variable_encode`)
- Added a `use_datetime` option to `TimeValidator`, which will cause it to use `datetime.time` objects instead of tuples. It was previously able to consume but not produce these objects.
- Added `<form:iferror name="not field_name">` when you want to include text only when a field has no errors.
- There was a problem installing 0.5.1 on Windows with Python 2.5, now resolved.

2.4.13 0.5.1

- Fixed compound validators and `not_empty` (was breaking SQLObject's PickleCol)

2.4.14 0.5

- Added `htmlfill.default_formatter_dict`, and you can poke new formatters in there to effective register them.
- Added an `escapenl` formatter (nl=newline) that escapes HTML and turns newlines into `<br>`.
- When `not_empty=False`, empty is assumed to be allowed. Thus `Int().to_python(None)` will now return `None`.

2.4.15 0.4

- Fixed up all the documentation.
- Validator `__doc__` attributes will include some automatically-appended information about all the message strings that validator uses.
- Deprecated `formencode.htmlform` module, because it is dumb.
- Added an `.all_messages()` method to all validators, primarily intended to be used for documentation purposes.
- Changed preferred name of `StringBoolean` to `StringBool` (to go with `bool` and `validators.Bool`). Old alias still available.
- Added `today_or_after` option to `validators.DateValidator`.
- Added a `validators.FileUploadKeeper` validator for helping with file uploads in failed forms. It still requires some annoying fiddling to make work, though, since file upload fields are so weird.
- Added `text_as_default` option to htmlfill. This treats all `<input type="something-weird">` elements as text fields. WHAT-WG adds weird input types, which can usually be usefully treated as text fields.
- Make all validators accept empty values if `not_empty` is False (the default). "Empty" means `""` or `None`, and will generally be converted None.
- Added `accept_python` boolean to all `FancyValidator` validators (which is most validators). This is a fixed version of the broken `validate_python` boolean added in 0.3. Also, it defaults to true, which means that all validators will not validate during `.from_python()` calls by default.
- Added `htmlfill.render(form, defaults, errors)` for easier rendering of forms.
- Errors automatically inserted by `htmlfill` will go at the top of the form if there's no field associated with the error (raised an error in 0.3).
- Added `formencode.sqlschema` for wrapping SQLObject classes/instances. See the docstring for more.
- Added `ignore_key_missing` to `Schema` objects, which ignore missing keys (where fields are present) when no `if_missing` is provided for the field.
- Renamed `validators.StateProvince.extraStates` to `extra_states`, to normalize style.

Bugfixes

- When checking destinations, `validators.URL` now allows redirect codes, and catches socket errors and turns them into proper errors.
- Fix typo in `htmlfill`
- Made URL and email regular expressions a little more lax/correct.
- A bunch of fixes to `validators.SignedString`, which apparently was completely broken.

2.4.16 0.3

- Allow errors to be inserted automatically into a form when using `formencode.htmlfill`, when a `<form:error>` tag isn't found for an error.
- Added `if_key_missing` attribute to `schema.Schema`, which will fill in any keys that are missing and pass them to the validator.
- `FancyValidator` has changed, adding `if_invalid_python` and `validate_python` options (which also apply to all subclasses). Also `if_empty` only applies to `to_python` conversions.
- `FancyValidator` now has a `strip` option, which if true and if input is a string, will strip whitespace from the string.
- Allow chained validators to validate otherwise-invalid forms, if they define a `validate_partial` method. The credit card validator does this.
- Handle `FieldStorage` input (from file uploads); added a `formencode.fieldstorage` module to wrap those instances in something a bit nicer. Added `validators.FieldStorageUploadConverter` to make this conversion.
- Added `StringBoolean` converter, which converts strings like `"true"` to Python booleans.

Bugfixes

- A couple fixes to `DateConverter`, `FieldsMatch`, `StringBoolean`, `CreditCardValidator`.
- Added missing `Validator.assert_string` method.
- `formencode.htmlfill_schemabuilder` handles checkboxes better.
- Be a little more careful about how `Invalid` exceptions are created (catch some errors sooner).
- Improved handling of non-string input in `htmlfill`.

Experiments

- Some experimental work in `formencode.formgen`. Experimental, I say!
- Added an experimental `formencode.context` module for dynamically-scoped variables.

CHAPTER 3

Documentation

3.1 FormEncode Validation

author Ian Bicking <ianb@colorstudy.com>

version 0.0.0

date April 21, 2023

Contents

3.1.1 Introduction

Validation (which encompasses conversion as well) is the core function of FormEncode. FormEncode really tries to *encode* the values from one source into another (hence the name). So a Python structure can be encoded in a series of HTML fields (a flat dictionary of strings). A HTML form submission can in turn be turned into a the original Python structure.

3.1.2 Using Validation

In FormEncode validation and conversion happen simultaneously. Frequently you cannot convert a value without ensuring its validity, and validation problems can occur in the middle of conversion.

The basic metaphor for validation is **to_python** and **from_python**. In this context "Python" is meant to refer to "here" – the trusted application, your own Python objects. The "other" may be a web form, an external database, an XML-RPC request, or any data source that is not completely trusted or does not map directly to Python's object model. `to_python()` is the process of taking external data and preparing it for internal use, `from_python()` generally reverses this process (`from_python()` is usually the less interesting of the pair, but provides some important features).

The core of this validation process is two methods and an exception:

```
>>> import formencode
>>> from formencode import validators
>>> validator = validators.Int()
>>> validator.to_python("10")
10
>>> validator.to_python("ten")
Traceback (most recent call last):
    ...
Invalid: Please enter an integer value
```

`"ten"` isn't a valid integer, so we get a `formencode.Invalid` exception. Typically we'd catch that exception, and use it for some sort of feedback. Like:

```
>>> def get_integer():
...     while 1:
...         try:
...             value = raw_input('Enter a number: ')
...             return validator.to_python(value)
...         except formencode.Invalid as e:
...             print (e)
...
>>> get_integer()
Enter a number: ten
Please enter an integer value
Enter a number: 10
10
```

We can also generalize this kind of function:

```
>>> def valid_input(prompt, validator):
...     while 1:
...         try:
...             value = raw_input(prompt)
...             return validator.to_python(value)
...         except formencode.Invalid as e:
```

(continues on next page)

(continued from previous page)

```
...             print (e)
>>> valid_input('Enter your email: ', validators.Email())
Enter your email: bob
An email address must contain a single @
Enter your email: bob@nowhere.com
'bob@nowhere.com'
```

`Invalid` exceptions generally give a good, user-readable error message about the problem with the input. Using the exception gets more complicated when you use compound data structures (dictionaries and lists), which we'll talk about *later*.

We'll talk more about these individual validators later, but first we'll talk about more complex validation than just integers or individual values.

Available Validators

There's lots of validators. The best way to read about the individual validators available in the *formencode.validators* module is to read about *validators* and *national*.

Compound Validators

While validating single values is useful, it's only a *little* useful. Much more interesting is validating a set of values. This is called a *Schema*.

For instance, imagine a registration form for a website. It takes the following fields, with restrictions:

- `first_name` (not empty)
- `last_name` (not empty)
- `email` (not empty, valid email)
- `username` (not empty, unique)
- `password` (reasonably secure)
- `password_confirm` (matches password)

There's a couple validators that aren't part of FormEncode, because they'll be specific to your application:

```
>>> # We don't really have a database of users, so we'll fake it:
>>> usernames = []
>>> class UniqueUsername(formencode.FancyValidator):
...     def _convert_to_python(self, value, state):
...         if value in usernames:
...             raise formencode.Invalid(
...                 'That username already exists',
...                 value, state)
...         return value
```

Note: The class `formencode.FancyValidator` is the superclass for most validators in FormEncode, and it provides a number of useful features that most validators can use – for instance, you can pass `strip=True` into any of these validators, and they'll strip whitespace from the incoming value before any other validation.

This overrides the internal `_convert_to_python()` method: `formencode.FancyValidator` adds a number of extra features, and then calls the internal `_convert_to_python()` method, which is the method you'll typically write. Contrary to the external method `to_python()`, its only concern is the conversion part, not the validation part. If further validation is necessary, this can be done in two other internal methods, either `_validate_python()` or `_validate_other()`. We will give an example for that later. When a validator finds an error, it raises an exception (`formencode.Invalid`), with the error message and the value and "state" objects. We'll talk about *state* later. Here's the other custom validator, that checks passwords against words in the standard Unix word file:

```
>>> class SecurePassword(formencode.FancyValidator):
...     words_filename = '/usr/share/dict/words'
...     def _convert_to_python(self, value, state):
...         f = open(self.words_filename)
...         lower = value.strip().lower()
...         for line in f:
...             if line.strip().lower() == lower:
...                 raise formencode.Invalid(
...                     'Please do not base your password on a '
...                     'dictionary term', value, state)
...         return value
```

And here's a schema:

```
>>> class Registration(formencode.Schema):
...     first_name = validators.ByteString(not_empty=True)
...     last_name = validators.ByteString(not_empty=True)
...     email = validators.Email(resolve_domain=True)
...     username = formencode.All(validators.PlainText(),
...                               UniqueUsername())
...     password = SecurePassword()
...     password_confirm = validators.ByteString()
...     chained_validators = [validators.FieldsMatch(
...         'password', 'password_confirm')]
```

Like any other validator, a `Registration` instance will have the `to_python()` and `from_python()` methods. The input should be a dictionary (or a Paste MultiDict), with keys like `"first_name"`, `"password"`, etc. The validators you give as attributes will be applied to each of the values of the dictionary. *All* the values will be validated, so if there are multiple invalid fields you will get information about all of them.

Most validators (anything that subclasses `formencode.FancyValidator`) will take a certain standard set of constructor keyword arguments. See *`formencode.api.FancyValidator`* for more – here we use `not_empty=True`.

Another notable validator is *`formencode.compound.All`* – this is a *compound validator* – that is, it's a validator that takes validators as input. Schemas are one example; in this case `All` takes a list of validators and applies each of them in turn. *`formencode.compound.Any`* is its compliment, that uses the first passing validator in its list.

`chained_validators` are validators that are run on the entire dictionary after other validation is done (`pre_validators` are applied before the schema validation). chained_validators will also allow for multiple validators to fail and report to the error_dict so, for example, if you have an email_confirm and a password_confirm fields and use FieldsMatch on both of them as follows:

```
>>> chained_validators = [
...     validators.FieldsMatch('password',
...                            'password_confirm'),
...     validators.FieldsMatch('email',
...                            'email_confirm')]
```

This will leave the error_dict with both password_confirm and email_confirm error keys, which is likely the desired

behavior for web forms.

Since a *formencode.schema.Schema* is just another kind of validator, you can nest these indefinitely, validating dictionaries of dictionaries.

Another way to do simple validation of a complete form is with `formencode.schema.SimpleFormValidator`. This class wraps a simple function that you write. For example:

```
>>> from formencode.schema import SimpleFormValidator
>>> def validate_state(value_dict, state, validator):
...     if value_dict.get('country', 'US') == 'US':
...         if not value_dict.get('state'):
...             return {'state': 'You must enter a state'}
>>> ValidateState = SimpleFormValidator(validate_state)
>>> ValidateState.to_python({'country': 'US'}, None)
Traceback (most recent call last):
    ...
Invalid: state: You must enter a state
```

The `validate_state()` function (or any validation function) returns any errors in the form (or it may raise Invalid directly). It can also modify the `value_dict` dictionary directly. When it returns None this indicates that everything is valid. You can use this with a `Schema` by putting `ValidateState` in `pre_validators` (all validation will be done before the schema's validation, and if there's an error the schema won't be run). Or you can put it in `chained_validators` and it will be run *after* the schema. If the schema fails (the values are invalid) then `ValidateState` will not be run, unless you set `validate_partial_form` to True (like `ValidateState = SimpleFormValidator(validate_state, validate_partial_form=True)`. If you validate a partial form you should be careful that you handle missing keys and other possibly-invalid values gracefully.

You can also validate lists of items using *formencode.foreach.ForEach*. For example, let's say we have a form where someone can edit a list of book titles. Each title has an associated book ID, so we can match up the new title and the book it is for:

```
>>> class BookSchema(formencode.Schema):
...     id = validators.Int()
...     title = validators.ByteString(not_empty=True)
>>> validator = formencode.ForEach(BookSchema())
```

The `validator` we've created will take a list of dictionaries as input (like `[{"id": "1", "title": "War & Peace"}, {"id": "2", "title": "Brave New World"}, ...]`). It applies the `BookSchema` to each entry, and collects any errors and reraises them. Of course, when you are validating input from an HTML form you won't get well structured data like this (we'll talk about that *later*).

Writing Your Own Validator

We gave a brief introduction to creating a validator earlier (`UniqueUsername` and `SecurePassword`). We'll discuss that a little more. Here's a more complete implementation of `SecurePassword`:

```
>>> import re
>>> class SecurePassword(validators.FancyValidator):
...
...     min = 3
...     non_letter = 1
...     letter_regex = re.compile(r'[a-zA-Z]')
...
...     messages = {
...         'too_few': 'Your password must be longer than %(min)i '
...                    'characters long',
```

(continues on next page)

(continued from previous page)

```
...         'non_letter': 'You must include at least %(non_letter)i '
...                       'characters in your password',
...         }
...
...     def _convert_to_python(self, value, state):
...         # _convert_to_python gets run before _validate_python.
...         # Here we strip whitespace off the password, because leading
...         # and trailing whitespace in a password is too elite.
...         return value.strip()
...
...     def _validate_python(self, value, state):
...         if len(value) < self.min:
...             raise formencode.Invalid(self.message("too_few", state,
...                                                   min=self.min),
...                                      value, state)
...         non_letters = self.letter_regex.sub('', value)
...         if len(non_letters) < self.non_letter:
...             raise formencode.Invalid(self.message("non_letter", state,
...                                                   non_letter=self.non_letter),
...                                      value, state)
```

With all validators, any arguments you pass to the constructor will be used to set instance variables. So `SecureValidator(min=5)` will be a minimum-five-character validator. This makes it easy to also subclass other validators, giving different default values.

Unlike the previous implementation we use the already mentioned `_validate_python()` method, which is another internal method `FancyValidator` allows us to override. `_validate_python()` doesn't have any return value, it simply raises an exception if it needs to. It validates the value *after* it has been converted (by `_convert_to_python()`). `_validate_other()` validates before conversion, but that's usually not that useful. The external method `to_python()` cares about the extra features such as the `if_empty` parameter, and uses the internal methods to do the actual conversion and validation; first it calls `_validate_other()`, then `_convert_to_python()` and at last `_validate_python()`.

The use of `self.message(...)` is meant to make the messages easy to format for different environments, and replacable (with translations, or simply with different text). Each message should have an identifier (`"min"` and `"non_letter"` in this example). The keyword arguments to `message()` are used for message substitution. See *Messages* for more.

Other Validator Usage

Validators use instance variables to store their customization information. You can use either subclassing or normal instantiation to set these. These are (effectively) equivalent:

```
>>> plain = validators.Regex(regex='^[a-zA-Z]+$')
>>> # and...
>>> class Plain(validators.Regex):
...     regex = '^[a-zA-Z]+$'
>>> plain = Plain()
```

You can actually use classes most places where you could use an instance; `to_python()` and `from_python()` will create instances as necessary, and many other methods are available on both the instance and the class level.

When dealing with nested validators this class syntax is often easier to work with, and better displays the structure.

There are several options that most validators support (including your own validators, if you subclass from `formencode.FancyValidator`):

if_empty: If set, then this value will be returned if the input evaluates to false (empty list, empty string, None, etc), but not the 0 or False objects. This only applies to `.to_python()`.

not_empty: If true, then if an empty value is given raise an error. (Both with `.to_python()` and also `.from_python()` if `.validate_python` is true).

strip: If true and the input is a string, strip it (occurs before empty tests).

if_invalid: If set, then when this validator would raise Invalid during `.to_python()`, instead return this value.

if_invalid_python: If set, when the Python value (converted with `.from_python()`) is invalid, this value will be returned.

accept_python: If True (the default), then `._validate_python()` and `._validate_other()` will not be called when `.from_python()` is used.

if_missing: Typically when a field is missing the schema will raise an error. In that case no validation is run – so things like `if_invalid` won't be triggered. This special attribute (if set) will be used when the field is missing, and no error will occur. (`None` or `()` are common values)

State

All the validators receive a magic, somewhat meaningless `state` argument (which defaults to `None`). It's used for very little in the validation system as distributed, but is primarily intended to be an object you can use to hook your validator into the context of the larger system.

For instance, imagine a validator that checks that a user is permitted access to some resource. How will the validator know which user is logged in? State! Imagine you are localizing it, how will the validator know the locale? State! Whatever else you need to pass in, just put it in the state object as an attribute, then look for that attribute in your validator.

Also, during compound validation (a *formencode.schema.Schema* or *formencode.foreach.ForEach*) the state (if not None) will have more instance variables added to it. During a `Schema` (dictionary) validation the instance variable `key` and `full_dict` will be added – `key` is the current key (i.e., validator name), and `full_dict` is the rest of the values being validated. During a `ForEeach` (list) validation, `index` and `full_list` will be set.

Invalid Exceptions

Besides the string error message, `formencode.Invalid` exceptions have a few other instance variables:

value: The input to the validator that failed.

state: The associated *state*.

msg: The error message (`str(exc)` returns this)

error_list: If the exception happened in a `ForEach` (list) validator, then this will contain a list of `Invalid` exceptions. Each item from the list will have an entry, either None for no error, or an exception.

error_dict: If the exception happened in a `Schema` (dictionary) validator, then this will contain `Invalid` exceptions for each failing field. Passing fields not be included in this dictionary.

unpack_errors(): This method returns a set of lists and dictionaries containing strings, for each error. It's an unpacking of `error_list`, `error_dict` and `msg`. If you get an Invalid exception from a `Schema`, you probably want to call this method on the exception object.

Messages, Language Customization

All of the error messages can be customized. Each error message has a key associated with it, like `"too_few"` in the registration example. You can overwrite these messages by using you own `messages = {"key": "text"}` in the class statement, or as an argument when you call a class. Either way, you do not lose messages that you do not define, you only overwrite ones that you specify.

Messages often take arguments, like the number of characters, the invalid portion of the field, etc. These are always substituted as a dictionary (by name). So you will use placeholders like `%(key)s` for each substitution. This way you can reorder or even ignore placeholders in your new message.

When you are creating a validator, for maximum flexibility you should use the `message()` method, like:

```
messages = {
    'key': 'my message (with a %(substitution)s)',
    }

def _validate_python(self, value, state):
    raise formencode.Invalid(self.message('key', state, substitution='apples'),
                             value, state)
```

Localization of Error Messages (i18n)

When a failed validation occurs FormEncode tries to output the error message in the appropriate language. For this it uses the standard gettext mechanism of python. To translate the message in the appropriate message FormEncode has to find a gettext function that translates the string. The language to be translated into and the used domain is determined by the found gettext function. To serve a standard translation mechanism and to enable custom translations it looks in the following order to find a gettext (_) function:

1. method of the `state` object
2. function `__builtin__._()`. This function is only used when:

   ```
   Validator.use_builtin_gettext == True #True is default
   ```

3. formencode builtin `_stdtrans()` function

   for standalone use of FormEncode. The language to use is determined out of the locale system (see gettext documentation). Optionally you can also set the language or the domain explicitly with the function:

   ```
   formencode.api.set_stdtranslation(domain="FormEncode", languages=["de"])
   ```

   Formencode comes with a Domain `FormEncode` and the corresponding messages in the directory `localedir/language/LC_MESSAGES/FormEncode.mo`

4. Custom gettext function and addtional parameters

   If you use a custom gettext function and you want FormEncode to call your function with additional parameters you can set the dictionary:

   ```
   Validators.gettextargs
   ```

Available languages

All available languages are distributed with the code. You can see the currently available languages in the source under the directory `formencode/i18n`.

If your language is not present yet, please consider contributing a translation (where <lang> is you language code):

```
$ svn co http://svn.formencode.org/FormEncode/trunk/
$ easy_install Babel
$ python setup.py init_catalog -l <lang>
$ emacs formencode/i18n/<lang>/LC_MESSAGES/FormEncode.po # or whatever
  # editor you prefer make the translation
$ python setup.py compile_catalog -l <lang>
```

Then test, and send the PO and MO files to g...@gregor-horvath.com.

See also the Python internationalization documents.

Optionally you can also add a test of your language to `tests/test_i18n.py`. An Example of a language test:

```
ne = formencode.validators.NotEmpty()
[...]
def test_de():
    _test_lang("de", u"Bitte einen Wert eingeben")
```

And the test for your language:

```
def test_<lang>():
    _test_lang("<lang>", u"<translation of Not Empty Text in the language <lang>")
```

HTTP/HTML Form Input

The validation expects nested data structures; specifically *formencode.schema.Schema* and *formencode.foreach.ForEach* deal with these structures well. HTML forms, however, do not produce nested structures – they produce flat structures with keys (input names) and associated values.

Validator includes the module *formencode.variabledecode*, which allows you to encode nested dictionary and list structures into a flat dictionary.

To do this it uses keys with `"."` for nested dictionaries, and `"-int"` for (ordered) lists. So something like:

key	value
names-1.fname	John
names-1.lname	Doe
names-2.fname	Jane
names-2.lname	Brown
names-3	Tim Smith
action	save
action.option	overwrite
action.confirm	yes

Will be mapped to:

```
{'names': [{'fname': "John", 'lname': "Doe"},
           {'fname': "Jane", 'lname': 'Brown'},
           "Tim Smith"],
 'action': {None: "save",
            'option': "overwrite",
            'confirm': "yes"},
}
```

In other words, `'a.b'` creates a dictionary in `'a'`, with `'b'` as a key (and if `'a'` already had a value, then that value is associated with the key `None`). Lists are created with keys with `'-int'`, where they are ordered by the integer (the integers are used for sorting, missing numbers in a sequence are ignored).

formencode.variabledecode.NestedVariables is a validator that decodes and encodes dictionaries using this algorithm. You can use it with a Schema's *pre_validators* attribute.

Of course, in the example we use the data is rather eclectic – for instance, Tim Smith doesn't have his name separated into first and last. Validators work best when you keep lists homogeneous. Also, it is hard to access the `'action'` key in the example; storing the options (action.option and action.confirm) under another key would be preferable.

3.2 htmlfill

author Ian Bicking <ianb@colorstudy.com>

Contents

3.2.1 Introduction

formencode.htmlfill is a library to fill out forms, both with default values and error messages. It's like a template library, but more limited, and it can be used with the output from other templates. It has no prerequesites, and can be used without any other parts of FormEncode.

3.2.2 Usage

The basic usage is something like this:

```
>>> from formencode import htmlfill
>>> form = '<input type="text" name="fname">'
>>> defaults = {'fname': 'Joe'}
>>> htmlfill.render(form, defaults)
'<input type="text" name="fname" value="Joe">'
```

The parser looks for HTML input elements (including `select` and `textarea`) and fills in the defaults. The quintessential way to use this would be with a form submission that had errors – you can return the form to the user with the values they entered, in addition to errors.

See *formencode.htmlfill.render()* for more.

Errors

Since errors are a common issue, this also has some features for filling the form with error messages. It defines two special tags for this purpose:

<form:error name="field_name" format="formatter">: This tag is eliminated completely if there is no error for the named field. Otherwise the error is passed through the given formatter (`"default"` if no `format` attribute is given).

<form:iferror name="field_name">...</form:iferror>: If the named field doesn't have an error, everything between the tags will be eliminated. Use `name="not field_name"` to invert the behavior (i.e., include text only if there are no errors for the field).

Formatters are functions that take the error text as a single argument. (In the future they may take extra arguments as well.) They return a string that is inserted into the template. By default, the formatter returns:

```
<span class="error-message">(message)</span><br>
```

In addition to explicit error tags, any leftover errors will be placed immediately above the associated input field.

The default formatters available to you:

default: HTML-quotes the error and wraps it in `<span class="error-message">`

none: Puts the error message in with no quoting of any kind. This allows you to put HTML in the error message, but might also expose you to cross-site scripting vulnerabilities.

escape: HTML-quotes the error, but doesn't wrap it in anything.

escapenl: HTML-quotes the error, and translates newlines to `<br>`

ignore: Swallows the error, emitting nothing. You can use this when you never want an error for a field to display.

Valid form templates

When you call `parser.close()` (also called by `render()`) the parser will check that you've fully used all the defaults and errors that were given in the constructor if you pass in `use_all_keys=True`. If there are leftover fields an `AssertionError` will be raised.

In most cases, htmlfill tries to keep the template the way it found it, without reformatting it too much. If HTMLParser chokes on the code, so will htmlfill.

3.3 Things To Do

version 0.0.0

date April 21, 2023

- Add a parallel to Pipe for the Any compound validator that performs validation for each validator to python from left to right.
- Make a test fixture for validators, to make testing really easy.
- Consider moving htmlfill to ElementTree or another DOM-ish structure, instead of HTMLParser. Or re-implement with another parser but same interface.
- Generate Javascript for validators, for client-side validation (when possible).
- Relatedly, test and give recipes for Ajax-ish validation, when fully client-side validation doesn't work.
- Better tests for `htmlfill` and `htmlfill_schemabuilder`.
- Include at least one good documented form generator. Consider including rich widgets (Javascript).
- Seperate out `doctest_xml_compare`, maybe (useful in any doctested web test).

- Make `doctest_xml_compare` work with wildcards/ellipses. Maybe with non-XHTML.
- Some more ways to build validation. Validation from docstrings or method signatures.

3.4 FormEncode Design

author Ian Bicking <ianb@colorstudy.com>

version 0.0.0

date April 21, 2023

Contents

This is a document to describe why FormEncode looks the way it looks, and how it fits into other applications. It also talks some about the false starts I've made.

3.4.1 Basic Metaphor

FormEncode performs look-before-you-leap validation. The idea being that you check all the data related to an operation, then apply it. This is in contrast to a transactional system, where you just start applying the data and if there's a problem you raise an exception. Someplace else you catch the exception and roll back the transaction. Of course FormEncode works fine with such a system, but because nothing is done until everything validates, you can use this without transactions.

FormEncode generally works on primitive types. These are things like strings, lists, dictionaries, integers, etc. This fits in with look-before-you-leap; often your domain objects won't exist until after you apply the user's request, so it's necessary to work on an early form of the data. Also, FormEncode doesn't know anything about your domain objects or classes; it's just easier to keep it this way.

Validation only operates on a single "value" at a time. This is Python, collections are easy, and collections are themselves a single "value" made up of many pieces. A "Schema validator" is a validator made up of many subvalidators. By using this single metaphor, without separating the concept of "field" and "form", it is possible to create reusable validators that work on compound structures, to validate "whole forms" instead of just single fields, and to support better validation composition.

Also, "validation" and "conversion" are generally applied at the same time. In the documentation this is frequently just referred to as "validation", but anywhere validation can happen, conversion can also happen.

3.4.2 Domain Objects

These are your objects, specific to your application. I know nothing about them, and cannot know. FormEncode doesn't do anything with these objects, and doesn't try to know anything about them.

3.4.3 Validation as directional, not intrinsic

One false start from earlier projects was an attempt to tie validators into the objects they validate against. E.g., you might have a SQLObject class like:

```
class Address(SQLObject):
    fname = StringCol(notNull=True)
    lname = StringCol(notNull=True)
    mi = StringCol()
```

It is tempting to take the restrictions of the `Address` class and automatically come up with a validation schema. This may yet be a viable goal (and to a degree is attainable), but in practical terms validation tends to be both more and less restrictive. Also, validation is contextual; what validation you apply is dependent on the source of the data.

Often in an API we are more restrictive than we may be in a user interface, demanding that everything be specified explicitly. In a UI we may assist the user by filling in values on their behalf. The specifics of this depend on the UI and the objects in question.

At the same time, we are often more restrictive in a UI. For instance, we may demand that the user enter something that appears to be a valid phone number. But for historical reasons, we may not make that demand for objects that already exist, or we may put in a tight restriction on the UI keeping in mind that it can more easily be relaxed and refined than a restriction in the domain objects or underlying database. Also, we may trust the programmer to use the API in a reasonable way, but we seldom trust user data in the same way.

In essence, there is an "inside" and an "outside" to the program. FormEncode is a toolkit for bridging those two areas in a sensible and secure way. The specific way we bridge this depends on the nature of the user interface. An XML-RPC interface can make some assumptions that a GUI cannot make. An HTML interface can typically make even fewer assumptions, including the basic integrity of the input data. It isn't reasonable that the object should know about all means of inputs, and the varying UI requirements of those inputs; user interfaces are volatile, and more art than science, but domain objects work better when they remain stable. For this reason the validation schemas are kept in separate objects.

It also didn't work well to annotate domain objects with validation schemas, though the option remains open. This is experimentation that belongs outside of the core of FormEncode, simply because it's more specific to your domain than it is to FormEncode.

3.4.4 Two sides, two aspects

FormEncode does both validation and conversion at the same time. Validation necessarily happens with every conversion; for instance, you may want to convert string representation of dates to internal date objects; that conversion can fail if the string representation is malformed.

To keep things simple, there's only one operation: conversion. An exception raised means there was an error. If you just want to validate, that's a conversion that doesn't change anything.

Similarly, there's two sides to the system, the foreign data and the local data. In Validator the local data is called "python" (meaning, a natural Python data structure), so we convert `to_python` and `from_python`. Unlike some systems, validators explicitly convert in *both* directions.

For instance, consider the date conversion. In one form, you may want a date like `mm/dd/yyyy`. It's easy enough to make the necessary converter; but the date object that the converter produces doesn't know how it's supposed to be

formatted for that form. Using `repr()` or *any* method that binds an object to its form representation is a bad idea. The converter best knows how to undo its work. So a date converter that expects `mm/dd/yyyy` will also know how to turn a datetime into that format.

(This becomes even more interesting with compound validators.)

3.4.5 Presentation

At one time FormEncode included form generation in addition to validation. The form generation worked okay; it was reasonably attractive, and in many ways quite powerful. I might revisit it. But generation is limited. It works *great* at first, then you hit a wall – you want to make a change, and you just *can't*, it doesn't fit into the automatic generation.

There are also many ways to approach the generation; again it's something that is tied to the framework, the presentation layer, and the domain objects, and FormEncode doesn't know anything about those.

FormEncode does provide htmlfill. *You* produce the form however you want. Write it out by hand. Use a templating language. Use a form generator. Whatever. Then htmlfill (which specifically understands HTML) fills in the form and any error messages. There are several advantages to this:

- Using `htmlfill`, form generation is easy. You can just think about how to map a form description or model class to simple HTML. You don't have to think about any of the low-level stuff about filling attributes with defaults or past request values.
- `htmlfill` works with anything that produces HTML. There's zero preference for any particular templating language, or even general style of templating language.
- If you do form generation, but it later turns out to be insufficiently flexible, you can put the generated form into your template and extend it there; you'll lose automatic synchronization with your models, but you won't lose any functionality.
- Hand-written forms are just as functional as generated forms.

3.4.6 Declarative and Imperative

All of the objects – schemas, repeating elements, individual validators – can be created imperatively, though more declarative styles often look better (specifically using subclassing instead of construction). You are free to build the objects either way.

An example of programmatically building form generation: `htmlfill_schemabuilder` looks for special attributes in an HTML form and builds a validation schema from that.

3.5 On Intentions And History

author Ian Bicking

I'm the author of FunFormKit, a form generation and validation package for Webware. I considered FunFormKit (FFK) to be a very powerful and complete package, with features that few other form validation packages for Python had (as to other languages, I haven't researched enough to know). It supported repeating and compound fields (which most packages do not), and had a very expressive validation system.

However, this is not FFK. In fact, it is a deprecation of FFK and does not provide backward compatibility. Why?

Probably the biggest problem was that FFK didn't support compound and repeating fields. Adding them made everything *much* more difficult – it was a sort of clever hack (maybe not even clever), and the result was very hard for anyone else to understand. Ultimately hard for me to understand.

Ontop of this was a structure that had too much coupling. Testing was difficult. I only came to like unit testing after FFK had gone through several revisions. FFK was not made with testability in mind. It can be hard to add later.

Also, I wanted to use pieces of FFK without the entire framework. Validation without the form generation was the biggest one. Alternate kinds of forms also interested me – making it easier to do highly granual templating, or non-HTML/HTTP forms. Alternate data sources, like SQL or XMLRPC, also seemed important. All of these were not easy within the interfaces that FFK used.

So... FormEncode! FormEncode takes a lot of ideas from FFK, and a lot of the code is just modified FFK code. All of it is reviewed and actively inserted into FormEncode, I'm not transferring anything wholesale.

3.6 FormEncode Internationalization (gettext)

author Gregor Horvath <gh at gregor-horvath dot com> 2006

There are different translation options available:

3.6.1 Domain "FormEncode"

for standalone use of FormEncode. The language to use is determined out of the local system (see gettext documentation http://docs.python.org/lib/node733.html). Optionally you can also set the language or the domain explicitly with the function.

example: formencode.api.set_stdtranslation(domain="FormEncode", languages=["de"])

The mo files are located in the i18n subdirectory of the formencode installation.

3.6.2 state._

A custom _ gettext function provided as attribute of the state object.

3.6.3 __builtins__._

A custom _ gettext function provided in the builtin namespace. This function is only used when:

Validator.use_builtin_gettext == True (True is default)

3.6.4 Without translation

If no translation mechanism is found a fallback returns the plain string.

3.7 Reference

3.7.1 `formencode.api` – Core classes for validation

Core classes for validation.

Module Contents

formencode.api.**is_empty**(*value*)
Check whether the given value should be considered "empty".

formencode.api.**is_validator**(*obj*)
Check whether obj is a Validator instance or class.

class formencode.api.**NoDefault**
A dummy value used for parameters with no default.

class formencode.api.**Invalid**(*msg*, *value*, *state*, *error_list=None*, *error_dict=None*)
This is raised in response to invalid input. It has several public attributes:

msg: The message, *without* values substituted. For instance, if you want HTML quoting of values, you can apply that.

substituteArgs: The arguments (a dictionary) to go with `msg`.

str(self): The message describing the error, with values substituted.

value: The offending (invalid) value.

state: The state that went with this validator. This is an application-specific object.

error_list: If this was a compound validator that takes a repeating value, and sub-validator(s) had errors, then this is a list of those exceptions. The list will be the same length as the number of values – valid values will have None instead of an exception.

error_dict: Like `error_list`, but for dictionary compound validators.

class formencode.api.**Validator**(**args*, ***kw*)
The base class of most validators. See `IValidator` for more, and `FancyValidator` for the more common (and more featureful) class.

Messages

formencode.api.**Identity**

class formencode.api.**FancyValidator**(**args*, ***kw*)
FancyValidator is the (abstract) superclass for various validators and converters. A subclass can validate, convert, or do both. There is no formal distinction made here.

Validators have two important external methods:

.to_python(value, state): Attempts to convert the value. If there is a problem, or the value is not valid, an Invalid exception is raised. The argument for this exception is the (potentially HTML-formatted) error message to give the user.

.from_python(value, state): Reverses `.to_python()`.

These two external methods make use of the following four important internal methods that can be overridden. However, none of these *have* to be overridden, only the ones that are appropriate for the validator.

._convert_to_python(value, state): This method converts the source to a Python value. It returns the converted value, or raises an Invalid exception if the conversion cannot be done. The argument to this exception should be the error message. Contrary to `.to_python()` it is only meant to convert the value, not to fully validate it.

._convert_from_python(value, state): Should undo `._convert_to_python()` in some reasonable way, returning a string.

._validate_other(value, state): Validates the source, before ._convert_to_python(), or after ._convert_from_python(). It's usually more convenient to use ._validate_python() however.

._validate_python(value, state): Validates a Python value, either the result of ._convert_to_python(), or the input to ._convert_from_python().

You should make sure that all possible validation errors are raised in at least one these four methods, not matter which.

Subclasses can also override the __init__() method if the declarative.Declarative model doesn't work for this.

Validators should have no internal state besides the values given at instantiation. They should be reusable and reentrant.

All subclasses can take the arguments/instance variables:

if_empty: If set, then this value will be returned if the input evaluates to false (empty list, empty string, None, etc), but not the 0 or False objects. This only applies to .to_python().

not_empty: If true, then if an empty value is given raise an error. (Both with .to_python() and also .from_python() if ._validate_python is true).

strip: If true and the input is a string, strip it (occurs before empty tests).

if_invalid: If set, then when this validator would raise Invalid during .to_python(), instead return this value.

if_invalid_python: If set, when the Python value (converted with .from_python()) is invalid, this value will be returned.

accept_python: If True (the default), then ._validate_python() and ._validate_other() will not be called when .from_python() is used.

These parameters are handled at the level of the external methods .to_python() and .from_python already; if you overwrite one of the internal methods, you usually don't need to care about them.

Messages

badType: The input must be a string (not a %(type)s: %(value)r)

empty: Please enter a value

noneType: The input must be a string (not None)

3.7.2 formencode.compound – Validate with multiple validators

Validators for applying validations in sequence.

Module Contents

class formencode.compound.**All**(**args*, ***kw*)

Check if all of the specified validators are valid.

This class is like an 'and' operator for validators. All validators must work, and the results are passed in turn through all validators for conversion in the order of evaluation. All is the same as *Pipe* but operates in the reverse order.

The order of evaluation differs depending on if you are validating to Python or from Python as follows:

The validators are evaluated right to left when validating to Python.

The validators are evaluated left to right when validating from Python.

Pipe is more intuitive when predominantly validating to Python.

Examples:

```
>>> from formencode.validators import DictConverter
>>> av = All(validators=[DictConverter({2: 1}),
... DictConverter({3: 2}), DictConverter({4: 3})])
>>> av.to_python(4)
1
>>> av.from_python(1)
4
```

Messages

badType: The input must be a string (not a %(type)s: %(value)r)

empty: Please enter a value

noneType: The input must be a string (not None)

class formencode.compound.**Any**(**args*, ***kw*)

Check if any of the specified validators is valid.

This class is like an 'or' operator for validators. The first validator/converter in the order of evaluation that validates the value will be used.

The order of evaluation differs depending on if you are validating to Python or from Python as follows:

The validators are evaluated right to left when validating to Python.

The validators are evaluated left to right when validating from Python.

Examples:

```
>>> from formencode.validators import DictConverter
>>> av = Any(validators=[DictConverter({2: 1}),
... DictConverter({3: 2}), DictConverter({4: 3})])
>>> av.to_python(3)
2
>>> av.from_python(2)
3
```

Messages

badType: The input must be a string (not a %(type)s: %(value)r)

empty: Please enter a value

noneType: The input must be a string (not None)

class formencode.compound.**Pipe**(**args*, ***kw*)

Pipe value through all specified validators.

This class works like *All* but the order of evaluation is opposite. All validators must work, and the results are passed in turn through each validator for conversion in the order of evaluation. A behaviour known to Unix and GNU users as 'pipe'.

The order of evaluation differs depending on if you are validating to Python or from Python as follows:

The validators are evaluated left to right when validating to Python.

The validators are evaluated right to left when validating from Python.

Examples:

```
>>> from formencode.validators import DictConverter
>>> pv = Pipe(validators=[DictConverter({1: 2}),
... DictConverter({2: 3}), DictConverter({3: 4})])
>>> pv.to_python(1)
4
>>> pv.from_python(4)
1
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

noneType: The input must be a string (not None)

3.7.3 `formencode.declarative` – Base class for Validators

Declarative objects for FormEncode.

Declarative objects have a simple protocol: you can use classes in lieu of instances and they are equivalent, and any keyword arguments you give to the constructor will override those instance variables. (So if a class is received, we'll simply instantiate an instance with no arguments).

You can provide a variable __unpackargs__ (a list of strings), and if the constructor is called with non-keyword arguments they will be interpreted as the given keyword arguments.

If __unpackargs__ is ('*', name), then all the arguments will be put in a variable by that name.

Also, you can define a __classinit__(cls, new_attrs) method, which will be called when the class is created (including subclasses).

Module Contents

class formencode.declarative.**Declarative**(**args*, ***kw*)

formencode.declarative.**classinstancemethod**(*func*)

Acts like a class method when called from a class, like an instance method when called by an instance. The method should take two arguments, 'self' and 'cls'; one of these will be None depending on how the method was called.

3.7.4 `formencode.doctest_xml_compare` – XML-based comparison of Doctest output

Module Contents

formencode.doctest_xml_compare.**xml_compare**(*x1*, *x2*, *reporter=None*)

formencode.doctest_xml_compare.**install**()

3.7.5 formencode.exc – Custom exceptions and warnings

Custom exceptions and warnings.

Module Contents

class formencode.exc.**FERuntimeWarning**
Run time warning.

3.7.6 formencode.foreach – Validate items in a list

Validator for repeating items.

Module Contents

class formencode.foreach.**ForEach**(**args*, ***kw*)
Use this to apply a validator/converter to each item in a list.

For instance:

```
ForEach(Int(), OneOf([1, 2, 3]))
```

Will take a list of values and try to convert each of them to an integer, and then check if each integer is 1, 2, or 3. Using multiple arguments is equivalent to:

```
ForEach(All(Int(), OneOf([1, 2, 3])))
```

Use convert_to_list=True if you want to force the input to be a list. This will turn non-lists into one-element lists, and None into the empty list. This tries to detect sequences by iterating over them (except strings, which aren't considered sequences).

ForEach will try to convert the entire list, even if errors are encountered. If errors are encountered, they will be collected and a single Invalid exception will be raised at the end (with error_list set).

If the incoming value is a set, then we return a set.

Messages

badType: The input must be a string (not a %(type)s: %(value)r)

empty: Please enter a value

noneType: The input must be a string (not None)

3.7.7 formencode.htmlrename – Rename fields in an HTML form

Module to rename form fields

Module Contents

formencode.htmlrename.**rename**(*form*, *rename_func*)
Rename all the form fields in the form (a string), using rename_func

rename_func will be called with one argument, the name of the field, and should return a new name.

formencode.htmlrename.**add_prefix**(*form*, *prefix*, *dotted=False*)

Add the given prefix to all the fields in the form.

If dotted is true, then add a dot between prefix and the previous name. Empty fields will use the prefix as the name (with no dot).

3.7.8 formencode.htmlfill – Fill in HTML forms

Parser for HTML forms, that fills in defaults and errors. See `render`.

Module Contents

formencode.htmlfill.**render**(*form*, *defaults=None*, *errors=None*, *use_all_keys=False*, *error_formatters=None*, *add_attributes=None*, *auto_insert_errors=True*, *auto_error_formatter=None*, *text_as_default=False*, *checkbox_checked_if_present=False*, *listener=None*, *encoding=None*, *error_class='error'*, *prefix_error=True*, *force_defaults=True*, *skip_passwords=False*, *data_formencode_form=None*, *data_formencode_ignore=None*)

Render the `form` (which should be a string) given the `defaults` and `errors`. Defaults are the values that go in the input fields (overwriting any values that are there) and errors are displayed inline in the form (and also effect input classes). Returns the rendered string.

If `auto_insert_errors` is true (the default) then any errors for which `<form:error>` tags can't be found will be put just above the associated input field, or at the top of the form if no field can be found.

If `use_all_keys` is true, if there are any extra fields from defaults or errors that couldn't be used in the form it will be an error.

`error_formatters` is a dictionary of formatter names to one-argument functions that format an error into HTML. Some default formatters are provided if you don't provide this.

`error_class` is the class added to input fields when there is an error for that field.

`add_attributes` is a dictionary of field names to a dictionary of attribute name/values. If the name starts with + then the value will be appended to any existing attribute (e.g., `{'+class': ' important'}`).

`auto_error_formatter` is used to create the HTML that goes above the fields. By default it wraps the error message in a span and adds a `<br>`.

If `text_as_default` is true (default false) then `<input type="unknown">` will be treated as text inputs.

If `checkbox_checked_if_present` is true (default false) then `<input type="checkbox">` will be set to `checked` if any corresponding key is found in the `defaults` dictionary, even a value that evaluates to False (like an empty string). This can be used to support pre-filling of checkboxes that do not have a `value` attribute, since browsers typically will only send the name of the checkbox in the form submission if the checkbox is checked, so simply the presence of the key would mean the box should be checked.

`listener` can be an object that watches fields pass; the only one currently is in `htmlfill_schemabuilder.SchemaBuilder`

`encoding` specifies an encoding to assume when mixing str and unicode text in the template.

`prefix_error` specifies if the HTML created by auto_error_formatter is put before the input control (default) or after the control.

`force_defaults` specifies if a field default is not given in the `defaults` dictionary then the control associated with the field should be set as an unsuccessful control. So checkboxes will be cleared, radio and select

controls will have no value selected, and textareas will be emptied. This defaults to `True`, which is appropriate the defaults are the result of a form submission.

`skip_passwords` specifies if password fields should be skipped when rendering form-content. If disabled the password fields will not be filled with anything, which is useful when you don't want to return a user's password in plain-text source.

`data_formencode_form` if a string is passed in (default *None*) only fields with the html attribute *data-formencode-form* that matches this string will be processed. For example: if a HTML fragment has two forms they can be differentiated to Formencode by decorating the input elements with attributes such as *data-formencode-form="a"* or *data-formencode-form="b"*, then instructing *render()* to only process the "a" or "b" fields.

`data_formencode_ignore` if True (default *None*) fields with the html attribute *data-formencode-ignore* will not be processed. This attribute need only be present in the tag: *data-formencode-ignore="1"*, *data-formencode-ignore=""* and *data-formencode-ignore* without a value are all valid signifiers.

class formencode.htmlfill.**htmlliteral**(*html*, *text=None*)

formencode.htmlfill.**default_formatter**(*error*)
Formatter that escapes the error, wraps the error in a span with class `error-message`, and adds a `<br>`

formencode.htmlfill.**escape_formatter**(*error*)
Formatter that escapes HTML, no more.

formencode.htmlfill.**escapenl_formatter**(*error*)
Formatter that escapes HTML, and translates newlines to `<br>`

formencode.htmlfill.**ignore_formatter**(*error*)
Formatter that emits nothing, regardless of the error.

formencode.htmlfill.**ignore_formatter**(*error*)
Formatter that emits nothing, regardless of the error.

class formencode.htmlfill.**FillingParser**(*defaults*, *errors=None*, *use_all_keys=False*, *error_formatters=None*, *error_class='error'*, *add_attributes=None*, *listener=None*, *auto_error_formatter=None*, *text_as_default=False*, *checkbox_checked_if_present=False*, *encoding=None*, *prefix_error=True*, *force_defaults=True*, *skip_passwords=False*, *data_formencode_form=None*, *data_formencode_ignore=None*)

Fills HTML with default values, as in a form.

Examples:

```
>>> defaults = dict(name='Bob Jones',
...                 occupation='Crazy Cultist',
...                 address='14 W. Canal\nNew Guinea',
...                 living='no',
...                 nice_guy=0)
>>> parser = FillingParser(defaults)
>>> parser.feed('''<input type="text" name="name" value="fill">
... <select name="occupation"> <option value="">Default</option>
... <option value="Crazy Cultist">Crazy cultist</option> </select>
... <textarea cols="20" style="width: 100%" name="address">
... An address</textarea>
... <input type="radio" name="living" value="yes">
```

(continues on next page)

(continued from previous page)

```
... <input type="radio" name="living" value="no">
... <input type="checkbox" name="nice_guy" checked="checked">''')
>>> parser.close()
>>> print (parser.text())
<input type="text" name="name" value="Bob Jones">
<select name="occupation">
<option value="">Default</option>
<option value="Crazy Cultist" selected="selected">Crazy cultist</option>
</select>
<textarea cols="20" style="width: 100%" name="address">14 W. Canal
New Guinea</textarea>
<input type="radio" name="living" value="yes">
<input type="radio" name="living" value="no" checked="checked">
<input type="checkbox" name="nice_guy">
```

3.7.9 `formencode.htmlfill_schemabuilder` – Read a Schema from an HTML Form

Extension to `htmlfill` that can parse out schema-defining statements.

You can either pass `SchemaBuilder` to `htmlfill.render` (the `listen` argument), or call `parse_schema` to just parse out a `Schema` object.

Module Contents

`formencode.htmlfill_schemabuilder.`**`parse_schema`**(*form*)

Given an HTML form, parse out the schema defined in it and return that schema.

class formencode.htmlfill_schemabuilder.**SchemaBuilder**(*validators={'_': <function <lambda>>, '__all__': ['Invalid', 'FancyValidator', 'Validator', 'ConfirmType', 'Wrapper', 'Constant', 'MaxLength', 'MinLength', 'NotEmpty', 'Empty', 'Regex', 'PlainText', 'OneOf', 'DictConverter', 'IndexListConverter', 'DateValidator', 'Bool', 'RangeValidator', 'Int', 'Number', 'ByteString', 'UnicodeString', 'String', 'Set', 'Email', 'URL', 'XRI', 'OpenId', 'FieldStorageUploadConverter', 'FileUploadKeeper', 'DateConverter', 'TimeConverter', 'ISODateTimeConverter', 'StripField', 'StringBool', 'StringBoolean', 'SignedString', 'IPAddress', 'CIDR', 'MACAddress', 'FormValidator', 'RequireIfMissing', 'RequireIfPresent', 'RequireIfMatching', 'FieldsMatch', 'CreditCardValidator', 'CreditCardExpires', 'CreditCardSecurityCode'], '__builtins__': {'ArithmeticError': <class 'ArithmeticError'>, 'AssertionError': <class 'AssertionError'>, 'AttributeError': <class 'AttributeError'>, 'BaseException': <class 'BaseException'>, 'BlockingIOError': <class 'BlockingIOError'>, 'BrokenPipeError': <class 'BrokenPipeError'>, 'BufferError': <class 'BufferError'>, 'BytesWarning': <class 'BytesWarning'>, 'ChildProcessError': <class 'ChildProcessError'>, 'ConnectionAbortedError': <class 'ConnectionAbortedError'>, 'ConnectionError': <class 'ConnectionError'>, 'ConnectionRefusedError': <class 'ConnectionRefusedError'>, 'ConnectionResetError': <class 'ConnectionResetError'>, 'DeprecationWarning': <class 'DeprecationWarning'>, 'EOFError': <class 'EOFError'>, 'Ellipsis': Ellipsis, 'EnvironmentError': <class 'OSError'>, 'Excep-*

3.7.10 formencode.htmlgen – Convenient building of ElementTree nodes

Kind of like htmlgen, only much simpler. The only important symbol that is exported is `html`.

This builds ElementTree nodes, but with some extra useful methods. (Open issue: should it use `ElementTree` more, and the raw `Element` stuff less?)

You create tags with attribute access. I.e., the `A` anchor tag is `html.a`. The attributes of the HTML tag are done with keyword arguments. The contents of the tag are the non-keyword arguments (concatenated). You can also use the special `c` keyword, passing a list, tuple, or single tag, and it will make up the contents (this is useful because keywords have to come after all non-keyword arguments, which is non-intuitive). Or you can chain them, adding the keywords with one call, then the body with a second call, like:

```
>>> str(html.a(href='http://yahoo.com')('<Yahoo>'))
'<a href="http://yahoo.com">&lt;Yahoo&gt;</a>'
```

Note that strings will be quoted; only tags given explicitly will remain unquoted.

If the value of an attribute is None, then no attribute will be inserted. So:

```
>>> str(html.a(href='http://www.yahoo.com', name=None,
...               c='Click Here'))
'<a href="http://www.yahoo.com">Click Here</a>'
```

If the value is None, then the empty string is used. Otherwise str() is called on the value.

`html` can also be called, and it will produce a special list from its arguments, which adds a `__str__` method that does `html.str` (which handles quoting, flattening these lists recursively, and using '' for `None`).

`html.comment` will generate an HTML comment, like `html.comment('comment text')` – note that it cannot take keyword arguments (because they wouldn't mean anything).

Examples:

```
>>> str(html.html(
...    html.head(html.title("Page Title")),
...    html.body(
...    bgcolor='#000066',
...    text='#ffffff',
...    c=[html.h1('Page Title'),
...       html.p('Hello world!')],
...    )))
'<html><head><title>Page Title</title></head><body bgcolor="#000066" text="#ffffff">
↪<h1>Page Title</h1><p>Hello world!</p></body></html>'
>>> str(html.a(href='#top')('return to top'))
'<a href="#top">return to top</a>'
```

Module Contents

class formencode.htmlgen.**_HTML**

class formencode.htmlgen.**Element**

3.7.11 formencode.national – Country specific validators

Country specific validators for use with FormEncode.

Contents

Module Contents

Note

To use CountryValidator and LanguageValidator, install either pycountry or TurboGears Version 1.x.

Country, State, and Postal Codes

class formencode.national.**DelimitedDigitsPostalCode**(*partition_lengths*, *delimiter=None*, *strict=False*, **args*, ***kw*)

Abstraction of common postal code formats, such as 55555, 55-555 etc. With constant amount of digits. By providing a single digit as partition you can obtain a trivial 'x digits' postal code validator.

For flexibility, input may use additional delimiters or delimters in a bad position. Only the minimum (or if strict, exact) number of digits has to be provided.

```
>>> german = DelimitedDigitsPostalCode(5)
>>> german.to_python('55555')
'55555'
>>> german.to_python('55 55-5')
'55555'
>>> german.to_python('5555')
Traceback (most recent call last):
    ...
Invalid: Please enter a zip code (5 digits)
>>> polish = DelimitedDigitsPostalCode([2, 3], '-')
>>> polish.to_python('55555')
'55-555'
>>> polish.to_python('55-555')
'55-555'
>>> polish.to_python('555-55')
'55-555'
>>> polish.to_python('5555')
Traceback (most recent call last):
    ...
Invalid: Please enter a zip code (nn-nnn)
>>> nicaragua = DelimitedDigitsPostalCode([3, 3, 1], '-')
>>> nicaragua.to_python('5554443')
'555-444-3'
>>> nicaragua.to_python('555-4443')
```

(continues on next page)

(continued from previous page)

```
'555-444-3'
>>> nicaragua.to_python('5555')
Traceback (most recent call last):
    ...
Invalid: Please enter a zip code (nnn-nnn-n)
```

Messages

badType: The input must be a string (not a %(type)s: %(value)r)

empty: Please enter a value

invalid: Please enter a zip code (%(format)s)

noneType: The input must be a string (not None)

formencode.national.**USPostalCode**(**args*, ***kw*)
US Postal codes (aka Zip Codes).

```
>>> uspc = USPostalCode()
>>> uspc.to_python('55555')
'55555'
>>> uspc.to_python('55555-5555')
'55555-5555'
>>> uspc.to_python('5555')
Traceback (most recent call last):
    ...
Invalid: Please enter a zip code (5 digits)
```

formencode.national.**GermanPostalCode**(**args*, ***kw*)

formencode.national.**FourDigitsPostalCode**(**args*, ***kw*)

formencode.national.**PolishPostalCode**(**args*, ***kw*)

class formencode.national.**ArgentinianPostalCode**(**args*, ***kw*)
Argentinian Postal codes.

```
>>> ArgentinianPostalCode.to_python('C1070AAM')
'C1070AAM'
>>> ArgentinianPostalCode.to_python('c 1070 aam')
'C1070AAM'
>>> ArgentinianPostalCode.to_python('5555')
Traceback (most recent call last):
    ...
Invalid: Please enter a zip code (LnnnnLLL)
```

Messages

badType: The input must be a string (not a %(type)s: %(value)r)

empty: Please enter a value

invalid: Please enter a zip code (%(format)s)

noneType: The input must be a string (not None)

class formencode.national.**CanadianPostalCode**(**args*, ***kw*)
Canadian Postal codes.

```
>>> CanadianPostalCode.to_python('V3H 1Z7')
'V3H 1Z7'
>>> CanadianPostalCode.to_python('v3h1z7')
'V3H 1Z7'
>>> CanadianPostalCode.to_python('5555')
Traceback (most recent call last):
    ...
Invalid: Please enter a zip code (LnL nLn)
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

invalid: Please enter a zip code (`%(format)s`)

noneType: The input must be a string (not None)

class formencode.national.**UKPostalCode**(**args*, ***kw*)

UK Postal codes. Please see BS 7666.

```
>>> UKPostalCode.to_python('BFPO 3')
'BFPO 3'
>>> UKPostalCode.to_python('LE11 3GR')
'LE11 3GR'
>>> UKPostalCode.to_python('l1a 3gr')
'L1A 3GR'
>>> UKPostalCode.to_python('5555')
Traceback (most recent call last):
    ...
Invalid: Please enter a valid postal code (for format see BS 7666)
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

invalid: Please enter a valid postal code (for format see BS 7666)

noneType: The input must be a string (not None)

class formencode.national.**CountryValidator**(**args*, ***kw*)

Will convert a country's name into its ISO-3166 abbreviation for unified storage in databases etc. and return a localized country name in the reverse step.

@See http://www.iso.org/iso/country_codes/iso_3166_code_lists.htm

```
>>> CountryValidator.to_python('Germany')
u'DE'
>>> CountryValidator.to_python('Finland')
u'FI'
>>> CountryValidator.to_python('UNITED STATES')
u'US'
>>> CountryValidator.to_python('Krakovia')
Traceback (most recent call last):
    ...
Invalid: That country is not listed in ISO 3166
>>> CountryValidator.from_python('DE')
u'Germany'
```

(continues on next page)

(continued from previous page)

```
>>> CountryValidator.from_python('FI')
u'Finland'
```

Messages

badType: The input must be a string (not a %(type)s: %(value)r)

empty: Please enter a value

noneType: The input must be a string (not None)

valueNotFound: That country is not listed in ISO 3166

class formencode.national.**PostalCodeInCountryFormat**(**args*, ***kw*)

Makes sure the postal code is in the country's format by chosing postal code validator by provided country code. Does convert it into the preferred format, too.

```
>>> fs = PostalCodeInCountryFormat('country', 'zip')
>>> sorted(fs.to_python(dict(country='DE', zip='30167')).items())
[('country', 'DE'), ('zip', '30167')]
>>> fs.to_python(dict(country='DE', zip='3008'))
Traceback (most recent call last):
    ...
Invalid: Given postal code does not match the country's format.
>>> sorted(fs.to_python(dict(country='PL', zip='34343')).items())
[('country', 'PL'), ('zip', '34-343')]
>>> fs = PostalCodeInCountryFormat('staat', 'plz')
>>> sorted(fs.to_python(dict(staat='GB', plz='l1a 3gr')).items())
[('plz', 'L1A 3GR'), ('staat', 'GB')]
```

Messages

badFormat: Given postal code does not match the country's format.

badType: The input must be a string (not a %(type)s: %(value)r)

empty: Please enter a value

noneType: The input must be a string (not None)

class formencode.national.**USStateProvince**(**args*, ***kw*)

Valid state or province code (two-letter).

Well, for now I don't know the province codes, but it does state codes. Give your own *states* list to validate other state-like codes; give *extra_states* to add values without losing the current state values.

```
>>> s = USStateProvince('XX')
>>> s.to_python('IL')
'IL'
>>> s.to_python('XX')
'XX'
>>> s.to_python('xx')
'XX'
>>> s.to_python('YY')
Traceback (most recent call last):
    ...
Invalid: That is not a valid state code
```

Messages

badType: The input must be a string (not a %(type)s: %(value)r)

empty: Please enter a state code

invalid: That is not a valid state code

noneType: The input must be a string (not None)

wrongLength: Please enter a state code with TWO letters

Phones and Addresses

class formencode.national.**USPhoneNumber**(**args*, ***kw*)

Validates, and converts to ###-###-####, optionally with extension (as ext.##...). Only support US phone numbers. See InternationalPhoneNumber for support for that kind of phone number.

```
>>> p = USPhoneNumber()
>>> p.to_python('333-3333')
Traceback (most recent call last):
    ...
Invalid: Please enter a number, with area code, in the form ###-###-####,␣
↪optionally with "ext.####"
>>> p.to_python('555-555-5555')
'555-555-5555'
>>> p.to_python('1-393-555-3939')
'1-393-555-3939'
>>> p.to_python('321.555.4949')
'321.555.4949'
>>> p.to_python('3335550000')
'3335550000'
```

Messages

badType: The input must be a string (not a %(type)s: %(value)r)

empty: Please enter a value

noneType: The input must be a string (not None)

phoneFormat: Please enter a number, with area code, in the form ###-###-####, optionally with "ext.####"

class formencode.national.**InternationalPhoneNumber**(**args*, ***kw*)

Validates, and converts phone numbers to +##-###-#######. Adapted from RFC 3966

@param default_cc country code for prepending if none is provided can be a paramerless callable

```
>>> c = InternationalPhoneNumber(default_cc=lambda: 49)
>>> c.to_python('0555/8114100')
'+49-555-8114100'
>>> p = InternationalPhoneNumber(default_cc=49)
>>> p.to_python('333-3333')
Traceback (most recent call last):
    ...
Invalid: Please enter a number, with area code, in the form +##-###-#######.
>>> p.to_python('0555/4860-300')
'+49-555-4860-300'
>>> p.to_python('0555-49924-51')
'+49-555-49924-51'
>>> p.to_python('0555 / 8114100')
'+49-555-8114100'
>>> p.to_python('0555/8114100')
```

(continues on next page)

(continued from previous page)

```
'+49-555-8114100'
>>> p.to_python('0555 8114100')
'+49-555-8114100'
>>> p.to_python(' +49 (0)555 350 60 0')
'+49-555-35060-0'
>>> p.to_python('+49 555 350600')
'+49-555-350600'
>>> p.to_python('0049/ 555/ 871 82 96')
'+49-555-87182-96'
>>> p.to_python('0555-2 50-30')
'+49-555-250-30'
>>> p.to_python('0555 43-1200')
'+49-555-43-1200'
>>> p.to_python('(05 55)4 94 33 47')
'+49-555-49433-47'
>>> p.to_python('(00 48-555)2 31 72 41')
'+48-555-23172-41'
>>> p.to_python('+973-555431')
'+973-555431'
>>> p.to_python('1-393-555-3939')
'+1-393-555-3939'
>>> p.to_python('+43 (1) 55528/0')
'+43-1-55528-0'
>>> p.to_python('+43 5555 429 62-0')
'+43-5555-42962-0'
>>> p.to_python('00 218 55 33 50 317 321')
'+218-55-3350317-321'
>>> p.to_python('+218 (0)55-3636639/38')
'+218-55-3636639-38'
>>> p.to_python('032 555555 367')
'+49-32-555555-367'
>>> p.to_python('(+86) 555 3876693')
'+86-555-3876693'
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

noneType: The input must be a string (not None)

phoneFormat: Please enter a number, with area code, in the form +##-###-#######.

Language Codes and Names

class formencode.national.**LanguageValidator**(**args*, ***kw*)

Converts a given language into its ISO 639 alpha 2 code, if there is any. Returns the language's full name in the reverse.

Warning: ISO 639 neither differentiates between languages such as Cantonese and Mandarin nor does it contain all spoken languages. E.g., Lechitic languages are missing. Warning: ISO 639 is a smaller subset of ISO 639-2

@param key_ok accept the language's code instead of its name for input defaults to True

```
>>> l = LanguageValidator()
>>> l.to_python('German')
```

(continues on next page)

(continued from previous page)

```
u'de'
>>> l.to_python('Chinese')
u'zh'
>>> l.to_python('Klingonian')
Traceback (most recent call last):
    ...
Invalid: That language is not listed in ISO 639
>>> l.from_python('de')
u'German'
>>> l.from_python('zh')
u'Chinese'
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

noneType: The input must be a string (not None)

valueNotFound: That language is not listed in ISO 639

3.7.12 `formencode.schema` – Validate complete forms

Module Contents

class formencode.schema.**Schema**(**args*, ***kw*)

A schema validates a dictionary of values, applying different validators (be key) to the different values. If allow_extra_fields=True, keys without validators will be allowed; otherwise they will raise Invalid. If filter_extra_fields is set to true, then extra fields are not passed back in the results.

Validators are associated with keys either with a class syntax, or as keyword arguments (class syntax is usually easier). Something like:

```
class MySchema(Schema):
    name = Validators.PlainText()
    phone = Validators.PhoneNumber()
```

These will not be available as actual instance variables, but will be collected in a dictionary. To remove a validator in a subclass that is present in a superclass, set it to None, like:

```
class MySubSchema(MySchema):
    name = None
```

Note that missing fields are handled at the Schema level. Missing fields can have the 'missing' message set to specify the error message, or if that does not exist the *schema* message 'missingValue' is used.

Messages

badDictType: The input must be dict-like (not a `%(type)s`: `%(value)r`)

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

missingValue: Missing value

noneType: The input must be a string (not None)

notExpected: The input field `%(name)s` was not expected.

singleValueExpected: Please provide only one value

3.7.13 `formencode.validators` – lots of useful validators

Validator/Converters for use with FormEncode.

Contents

Module Contents

Basic Types

class formencode.validators.**ByteString**(**args*, ***kw*)

Convert to byte string, treating empty things as the empty string.

Under Python 2.x you can also use the alias *String* for this validator.

Also takes a *max* and *min* argument, and the string length must fall in that range.

Also you may give an *encoding* argument, which will encode any unicode that is found. Lists and tuples are joined with *list_joiner* (default `', '`) in `from_python`.

```
>>> ByteString(min=2).to_python('a')
Traceback (most recent call last):
    ...
Invalid: Enter a value 2 characters long or more
>>> ByteString(max=10).to_python('xxxxxxxxxxx')
Traceback (most recent call last):
    ...
Invalid: Enter a value not more than 10 characters long
>>> ByteString().from_python(None)
''
>>> ByteString().from_python([])
''
>>> ByteString().to_python(None)
''
```

(continues on next page)

(continued from previous page)

```
>>> ByteString(min=3).to_python(None)
Traceback (most recent call last):
    ...
Invalid: Please enter a value
>>> ByteString(min=1).to_python('')
Traceback (most recent call last):
    ...
Invalid: Please enter a value
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

noneType: The input must be a string (not None)

tooLong: Enter a value not more than `%(max)i` characters long

tooShort: Enter a value `%(min)i` characters long or more

class formencode.validators.**StringBool**(**args*, ***kw*)

Converts a string to a boolean.

Values like 'true' and 'false' are considered True and False, respectively; anything in `true_values` is true, anything in `false_values` is false, case-insensitive). The first item of those lists is considered the preferred form.

```
>>> s = StringBool()
>>> s.to_python('yes'), s.to_python('no')
(True, False)
>>> s.to_python(1), s.to_python('N')
(True, False)
>>> s.to_python('ye')
Traceback (most recent call last):
    ...
Invalid: Value should be 'true' or 'false'
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

noneType: The input must be a string (not None)

string: Value should be `%(true)r` or `%(false)r`

class formencode.validators.**Bool**(**args*, ***kw*)

Always Valid, returns True or False based on the value and the existance of the value.

If you want to convert strings like `'true'` to booleans, then use `StringBool`.

Examples:

```
>>> Bool.to_python(0)
False
>>> Bool.to_python(1)
True
>>> Bool.to_python('')
False
```

(continues on next page)

(continued from previous page)

```
>>> Bool.to_python(None)
False
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

noneType: The input must be a string (not None)

class formencode.validators.**Int**(**args*, ***kw*)

Convert a value to an integer.

Example:

```
>>> Int.to_python('10')
10
>>> Int.to_python('ten')
Traceback (most recent call last):
    ...
Invalid: Please enter an integer value
>>> Int(min=5).to_python('6')
6
>>> Int(max=10).to_python('11')
Traceback (most recent call last):
    ...
Invalid: Please enter a number that is 10 or smaller
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

integer: Please enter an integer value

noneType: The input must be a string (not None)

tooHigh: Please enter a number that is `%(max)s` or smaller

tooLow: Please enter a number that is `%(min)s` or greater

class formencode.validators.**Number**(**args*, ***kw*)

Convert a value to a float or integer.

Tries to convert it to an integer if no information is lost.

Example:

```
>>> Number.to_python('10')
10
>>> Number.to_python('10.5')
10.5
>>> Number.to_python('ten')
Traceback (most recent call last):
    ...
Invalid: Please enter a number
>>> Number.to_python([1.2])
Traceback (most recent call last):
    ...
```

(continues on next page)

(continued from previous page)

```
Invalid: Please enter a number
>>> Number(min=5).to_python('6.5')
6.5
>>> Number(max=10.5).to_python('11.5')
Traceback (most recent call last):
    ...
Invalid: Please enter a number that is 10.5 or smaller
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

noneType: The input must be a string (not None)

number: Please enter a number

tooHigh: Please enter a number that is `%(max)s` or smaller

tooLow: Please enter a number that is `%(min)s` or greater

class formencode.validators.**UnicodeString**(***kw*)

Convert things to unicode string.

This is implemented as a specialization of the ByteString class.

Under Python 3.x you can also use the alias *String* for this validator.

In addition to the String arguments, an encoding argument is also accepted. By default the encoding will be utf-8. You can overwrite this using the encoding parameter. You can also set inputEncoding and outputEncoding differently. An inputEncoding of None means "do not decode", an outputEncoding of None means "do not encode".

All converted strings are returned as Unicode strings.

```
>>> UnicodeString().to_python(None) == u''
True
>>> UnicodeString().to_python([]) == u''
True
>>> UnicodeString(encoding='utf-7').to_python('Ni Ni Ni') == u'Ni Ni Ni'
True
```

Messages

badEncoding: Invalid data or incorrect encoding

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

noneType: The input must be a string (not None)

tooLong: Enter a value not more than `%(max)i` characters long

tooShort: Enter a value `%(min)i` characters long or more

class formencode.validators.**Set**(**args*, ***kw*)

This is for when you think you may return multiple values for a certain field.

This way the result will always be a list, even if there's only one result. It's equivalent to ForEach(convert_to_list=True).

If you give `use_set=True`, then it will return an actual `set` object.

```
>>> Set.to_python(None)
[]
>>> Set.to_python('this')
['this']
>>> Set.to_python(('this', 'that'))
['this', 'that']
>>> s = Set(use_set=True)
>>> s.to_python(None)
set([])
>>> s.to_python('this')
set(['this'])
>>> s.to_python(('this',))
set(['this'])
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

noneType: The input must be a string (not None)

Basic Validator/Converters

class formencode.validators.**ConfirmType**(**args*, ***kw*)

Confirms that the input/output is of the proper type.

Uses the parameters:

subclass: The class or a tuple of classes; the item must be an instance of the class or a subclass.

type: A type or tuple of types (or classes); the item must be of the exact class or type. Subclasses are not allowed.

Examples:

```
>>> cint = ConfirmType(subclass=int)
>>> cint.to_python(True)
True
>>> cint.to_python('1')
Traceback (most recent call last):
    ...
Invalid: '1' is not a subclass of <type 'int'>
>>> cintfloat = ConfirmType(subclass=(float, int))
>>> cintfloat.to_python(1.0), cintfloat.from_python(1.0)
(1.0, 1.0)
>>> cintfloat.to_python(1), cintfloat.from_python(1)
(1, 1)
>>> cintfloat.to_python(None)
Traceback (most recent call last):
    ...
Invalid: None is not a subclass of one of the types <type 'float'>, <type 'int'>
>>> cint2 = ConfirmType(type=int)
>>> cint2(accept_python=False).from_python(True)
Traceback (most recent call last):
    ...
Invalid: True must be of the type <type 'int'>
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

inSubclass: `%(object)r` is not a subclass of one of the types `%(subclassList)s`

inType: `%(object)r` must be one of the types `%(typeList)s`

noneType: The input must be a string (not None)

subclass: `%(object)r` is not a subclass of `%(subclass)s`

type: `%(object)r` must be of the type `%(type)s`

class formencode.validators.**Wrapper**(**args*, ***kw*)

Used to convert functions to validator/converters.

You can give a simple function for *_convert_to_python*, *_convert_from_python*, *_validate_python* or *_validate_other*. If that function raises an exception, the value is considered invalid. Whatever value the function returns is considered the converted value.

Unlike validators, the *state* argument is not used. Functions like *int* can be used here, that take a single argument.

Note that as Wrapper will generate a FancyValidator, empty values (those who pass `FancyValidator.is_empty`) will return `None`. To override this behavior you can use `Wrapper(empty_value=callable)`. For example passing `Wrapper(empty_value=lambda val: val)` will return the value itself when is considered empty.

Examples:

```
>>> def downcase(v):
...     return v.lower()
>>> wrap = Wrapper(convert_to_python=downcase)
>>> wrap.to_python('This')
'this'
>>> wrap.from_python('This')
'This'
>>> wrap.to_python('') is None
True
>>> wrap2 = Wrapper(
...     convert_from_python=downcase, empty_value=lambda value: value)
>>> wrap2.from_python('This')
'this'
>>> wrap2.to_python('')
''
>>> wrap2.from_python(1)
Traceback (most recent call last):
  ...
Invalid: 'int' object has no attribute 'lower'
>>> wrap3 = Wrapper(validate_python=int)
>>> wrap3.to_python('1')
'1'
>>> wrap3.to_python('a')
Traceback (most recent call last):
  ...
Invalid: invalid literal for int()...
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

noneType: The input must be a string (not None)

class formencode.validators.**Constant**(**args*, ***kw*)

This converter converts everything to the same thing.

I.e., you pass in the constant value when initializing, then all values get converted to that constant value.

This is only really useful for funny situations, like:

```
# Any evaluates sub validators in reverse order for to_python
fromEmailValidator = Any(
                       Constant('unknown@localhost'),
                          Email())
```

In this case, the if the email is not valid `'unknown@localhost'` will be used instead. Of course, you could use `if_invalid` instead.

Examples:

```
>>> Constant('X').to_python('y')
'X'
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

noneType: The input must be a string (not None)

class formencode.validators.**StripField**(**args*, ***kw*)

Take a field from a dictionary, removing the key from the dictionary.

`name` is the key. The field value and a new copy of the dictionary with that field removed are returned.

```
>>> StripField('test').to_python({'a': 1, 'test': 2})
(2, {'a': 1})
>>> StripField('test').to_python({})
Traceback (most recent call last):
    ...
Invalid: The name 'test' is missing
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

missing: The name `%(name)s` is missing

noneType: The input must be a string (not None)

class formencode.validators.**OneOf**(**args*, ***kw*)

Tests that the value is one of the members of a given list.

If `testValueList=True`, then if the input value is a list or tuple, all the members of the sequence will be checked (i.e., the input must be a subset of the allowed values).

Use `hideList=True` to keep the list of valid values out of the error message in exceptions.

Examples:

```
>>> oneof = OneOf([1, 2, 3])
>>> oneof.to_python(1)
1
>>> oneof.to_python(4)
Traceback (most recent call last):
  ...
Invalid: Value must be one of: 1; 2; 3 (not 4)
>>> oneof(testValueList=True).to_python([2, 3, [1, 2, 3]])
[2, 3, [1, 2, 3]]
>>> oneof.to_python([2, 3, [1, 2, 3]])
Traceback (most recent call last):
  ...
Invalid: Value must be one of: 1; 2; 3 (not [2, 3, [1, 2, 3]])
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

invalid: Invalid value

noneType: The input must be a string (not None)

notIn: Value must be one of: `%(items)s` (not `%(value)r`)

class formencode.validators.**DictConverter**(**args*, ***kw*)

Converts values based on a dictionary which has values as keys for the resultant values.

If `allowNull` is passed, it will not balk if a false value (e.g., '' or None) is given (it will return None in these cases).

to_python takes keys and gives values, from_python takes values and gives keys.

If you give hideDict=True, then the contents of the dictionary will not show up in error messages.

Examples:

```
>>> dc = DictConverter({1: 'one', 2: 'two'})
>>> dc.to_python(1)
'one'
>>> dc.from_python('one')
1
>>> dc.to_python(3)
Traceback (most recent call last):
    ....
Invalid: Enter a value from: 1; 2
>>> dc2 = dc(hideDict=True)
>>> dc2.hideDict
True
>>> dc2.dict
{1: 'one', 2: 'two'}
>>> dc2.to_python(3)
Traceback (most recent call last):
    ....
Invalid: Choose something
>>> dc.from_python('three')
Traceback (most recent call last):
    ....
Invalid: Nothing in my dictionary goes by the value 'three'.  Choose one of: 'one
↪'; 'two'
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

chooseKey: Enter a value from: `%(items)s`

chooseValue: Nothing in my dictionary goes by the value `%(value)s`. Choose one of: `%(items)s`

empty: Please enter a value

keyNotFound: Choose something

noneType: The input must be a string (not None)

valueNotFound: That value is not known

class formencode.validators.**IndexListConverter**(**args*, ***kw*)

Converts a index (which may be a string like '2') to the value in the given list.

Examples:

```
>>> index = IndexListConverter(['zero', 'one', 'two'])
>>> index.to_python(0)
'zero'
>>> index.from_python('zero')
0
>>> index.to_python('1')
'one'
>>> index.to_python(5)
Traceback (most recent call last):
Invalid: Index out of range
>>> index(not_empty=True).to_python(None)
Traceback (most recent call last):
Invalid: Please enter a value
>>> index.from_python('five')
Traceback (most recent call last):
Invalid: Item 'five' was not found in the list
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

integer: Must be an integer index

noneType: The input must be a string (not None)

notFound: Item `%(value)s` was not found in the list

outOfRange: Index out of range

Simple Validators

class formencode.validators.**MaxLength**(**args*, ***kw*)

Invalid if the value is longer than *maxLength*. Uses len(), so it can work for strings, lists, or anything with length.

Examples:

```
>>> max5 = MaxLength(5)
>>> max5.to_python('12345')
'12345'
```

(continues on next page)

(continued from previous page)

```
>>> max5.from_python('12345')
'12345'
>>> max5.to_python('123456')
Traceback (most recent call last):
  ...
Invalid: Enter a value less than 5 characters long
>>> max5(accept_python=False).from_python('123456')
Traceback (most recent call last):
  ...
Invalid: Enter a value less than 5 characters long
>>> max5.to_python([1, 2, 3])
[1, 2, 3]
>>> max5.to_python([1, 2, 3, 4, 5, 6])
Traceback (most recent call last):
  ...
Invalid: Enter a value less than 5 characters long
>>> max5.to_python(5)
Traceback (most recent call last):
  ...
Invalid: Invalid value (value with length expected)
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

invalid: Invalid value (value with length expected)

noneType: The input must be a string (not None)

tooLong: Enter a value less than `%(maxLength)i` characters long

class formencode.validators.**MinLength**(**args*, ***kw*)

Invalid if the value is shorter than *minlength*. Uses len(), so it can work for strings, lists, or anything with length. Note that you **must** use `not_empty=True` if you don't want to accept empty values – empty values are not tested for length.

Examples:

```
>>> min5 = MinLength(5)
>>> min5.to_python('12345')
'12345'
>>> min5.from_python('12345')
'12345'
>>> min5.to_python('1234')
Traceback (most recent call last):
  ...
Invalid: Enter a value at least 5 characters long
>>> min5(accept_python=False).from_python('1234')
Traceback (most recent call last):
  ...
Invalid: Enter a value at least 5 characters long
>>> min5.to_python([1, 2, 3, 4, 5])
[1, 2, 3, 4, 5]
>>> min5.to_python([1, 2, 3])
Traceback (most recent call last):
  ...
Invalid: Enter a value at least 5 characters long
```

(continues on next page)

(continued from previous page)

```
>>> min5.to_python(5)
Traceback (most recent call last):
  ...
Invalid: Invalid value (value with length expected)
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

invalid: Invalid value (value with length expected)

noneType: The input must be a string (not None)

tooShort: Enter a value at least `%(minLength)i` characters long

class formencode.validators.**NotEmpty**(**args*, ***kw*)

Invalid if value is empty (empty string, empty list, etc).

Generally for objects that Python considers false, except zero which is not considered invalid.

Examples:

```
>>> ne = NotEmpty(messages=dict(empty='enter something'))
>>> ne.to_python('')
Traceback (most recent call last):
  ...
Invalid: enter something
>>> ne.to_python(0)
0
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

noneType: The input must be a string (not None)

class formencode.validators.**Empty**(**args*, ***kw*)

Invalid unless the value is empty. Use cleverly, if at all.

Examples:

```
>>> Empty.to_python(0)
Traceback (most recent call last):
  ...
Invalid: You cannot enter a value here
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

noneType: The input must be a string (not None)

notEmpty: You cannot enter a value here

class formencode.validators.**Regex**(**args*, ***kw*)

Invalid if the value doesn't match the regular expression *regex*.

The regular expression can be a compiled re object, or a string which will be compiled for you.

Use strip=True if you want to strip the value before validation, and as a form of conversion (often useful).

Examples:

```
>>> cap = Regex(r'^[A-Z]+$')
>>> cap.to_python('ABC')
'ABC'
```

Note that `.from_python()` calls (in general) do not validate the input:

```
>>> cap.from_python('abc')
'abc'
>>> cap(accept_python=False).from_python('abc')
Traceback (most recent call last):
  ...
Invalid: The input is not valid
>>> cap.to_python(1)
Traceback (most recent call last):
  ...
Invalid: The input must be a string (not a <type 'int'>: 1)
>>> Regex(r'^[A-Z]+$', strip=True).to_python('  ABC  ')
'ABC'
>>> Regex(r'this', regexOps=('I',)).to_python('THIS')
'THIS'
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

invalid: The input is not valid

noneType: The input must be a string (not None)

class formencode.validators.**PlainText**(**args*, ***kw*)

Test that the field contains only letters, numbers, underscore, and the hyphen. Subclasses Regex.

Examples:

```
>>> PlainText.to_python('_this9_')
'_this9_'
>>> PlainText.from_python('  this  ')
'  this  '
>>> PlainText(accept_python=False).from_python('  this  ')
Traceback (most recent call last):
  ...
Invalid: Enter only letters, numbers, - (hyphen) or _ (underscore)
>>> PlainText(strip=True).to_python('  this  ')
'this'
>>> PlainText(strip=True).from_python('  this  ')
'this'
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

invalid: Enter only letters, numbers, - (hyphen) or _ (underscore)

noneType: The input must be a string (not None)

Dates and Times

class formencode.validators.**DateValidator**(**args*, ***kw*)

Validates that a date is within the given range. Be sure to call DateConverter first if you aren't expecting mxDateTime input.

`earliest_date` and `latest_date` may be functions; if so, they will be called each time before validating.

`after_now` means a time after the current timestamp; note that just a few milliseconds before now is invalid! `today_or_after` is more permissive, and ignores hours and minutes.

Examples:

```
>>> from datetime import datetime, timedelta
>>> d = DateValidator(earliest_date=datetime(2003, 1, 1))
>>> d.to_python(datetime(2004, 1, 1))
datetime.datetime(2004, 1, 1, 0, 0)
>>> d.to_python(datetime(2002, 1, 1))
Traceback (most recent call last):
    ...
Invalid: Date must be after Wednesday, 01 January 2003
>>> d.to_python(datetime(2003, 1, 1))
datetime.datetime(2003, 1, 1, 0, 0)
>>> d = DateValidator(after_now=True)
>>> now = datetime.now()
>>> d.to_python(now+timedelta(seconds=5)) == now+timedelta(seconds=5)
True
>>> d.to_python(now-timedelta(days=1))
Traceback (most recent call last):
    ...
Invalid: The date must be sometime in the future
>>> d.to_python(now+timedelta(days=1)) > now
True
>>> d = DateValidator(today_or_after=True)
>>> d.to_python(now) == now
True
```

Messages

after: Date must be after `%(date)s`

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

before: Date must be before `%(date)s`

date_format: %%A, %%d %%B %%Y

empty: Please enter a value

future: The date must be sometime in the future

noneType: The input must be a string (not None)

class formencode.validators.**DateConverter**(**args*, ***kw*)

Validates and converts a string date, like mm/yy, dd/mm/yy, dd-mm-yy, etc. Using `month_style` you can support the three general styles `mdy` = `us` = `mm/dd/yyyy`, `dmy` = `euro` = `dd/mm/yyyy` and `ymd` = `iso` = `yyyy/mm/dd`.

Accepts English month names, also abbreviated. Returns value as a datetime object (you can get mx.DateTime objects if you use `datetime_module='mxDateTime'`). Two year dates are assumed to be within 1950-2020, with dates from 21-49 being ambiguous and signaling an error.

Use accept_day=False if you just want a month/year (like for a credit card expiration date).

```
>>> d = DateConverter()
>>> d.to_python('12/3/09')
datetime.date(2009, 12, 3)
>>> d.to_python('12/3/2009')
datetime.date(2009, 12, 3)
>>> d.to_python('2/30/04')
Traceback (most recent call last):
    ...
Invalid: That month only has 29 days
>>> d.to_python('13/2/05')
Traceback (most recent call last):
    ...
Invalid: Please enter a month from 1 to 12
>>> d.to_python('1/1/200')
Traceback (most recent call last):
    ...
Invalid: Please enter a four-digit year after 1899
```

If you change `month_style` you can get European-style dates:

```
>>> d = DateConverter(month_style='dd/mm/yyyy')
>>> date = d.to_python('12/3/09')
>>> date
datetime.date(2009, 3, 12)
>>> d.from_python(date)
'12/03/2009'
```

Messages

badFormat: Please enter the date in the form `%(format)s`

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

dayRange: That month only has `%(days)i` days

empty: Please enter a value

fourDigitYear: Please enter a four-digit year after 1899

invalidDate: That is not a valid day (`%(exception)s`)

invalidDay: Please enter a valid day

invalidYear: Please enter a number for the year

monthRange: Please enter a month from 1 to 12

noneType: The input must be a string (not None)

unknownMonthName: Unknown month name: `%(month)s`

wrongFormat: Please enter the date in the form `%(format)s`

class formencode.validators.**TimeConverter**(**args*, ***kw*)

Converts times in the format HH:MM:SSampm to (h, m, s). Seconds are optional.

For ampm, set use_ampm = True. For seconds, use_seconds = True. Use 'optional' for either of these to make them optional.

Examples:

```
>>> tim = TimeConverter()
>>> tim.to_python('8:30')
(8, 30)
>>> tim.to_python('20:30')
(20, 30)
>>> tim.to_python('30:00')
Traceback (most recent call last):
    ...
Invalid: You must enter an hour in the range 0-23
>>> tim.to_python('13:00pm')
Traceback (most recent call last):
    ...
Invalid: You must enter an hour in the range 1-12
>>> tim.to_python('12:-1')
Traceback (most recent call last):
    ...
Invalid: You must enter a minute in the range 0-59
>>> tim.to_python('12:02pm')
(12, 2)
>>> tim.to_python('12:02am')
(0, 2)
>>> tim.to_python('1:00PM')
(13, 0)
>>> tim.from_python((13, 0))
'13:00:00'
>>> tim2 = tim(use_ampm=True, use_seconds=False)
>>> tim2.from_python((13, 0))
'1:00pm'
>>> tim2.from_python((0, 0))
'12:00am'
>>> tim2.from_python((12, 0))
'12:00pm'
```

Examples with `datetime.time`:

```
>>> v = TimeConverter(use_datetime=True)
>>> a = v.to_python('18:00')
>>> a
datetime.time(18, 0)
>>> b = v.to_python('30:00')
Traceback (most recent call last):
    ...
Invalid: You must enter an hour in the range 0-23
>>> v2 = TimeConverter(prefer_ampm=True, use_datetime=True)
>>> v2.from_python(a)
'6:00:00pm'
>>> v3 = TimeConverter(prefer_ampm=True,
...                    use_seconds=False, use_datetime=True)
>>> a = v3.to_python('18:00')
>>> a
datetime.time(18, 0)
>>> v3.from_python(a)
'6:00pm'
>>> a = v3.to_python('18:00:00')
Traceback (most recent call last):
    ...
Invalid: You may not enter seconds
```

Messages

badHour: You must enter an hour in the range `%(range)s`

badMinute: You must enter a minute in the range 0-59

badNumber: The `%(part)s` value you gave is not a number: `%(number)r`

badSecond: You must enter a second in the range 0-59

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

minutesRequired: You must enter minutes (after a :)

noAMPM: You must indicate AM or PM

noSeconds: You may not enter seconds

noneType: The input must be a string (not None)

secondsRequired: You must enter seconds

tooManyColon: There are too many :'s

HTML Form Helpers

class formencode.validators.**SignedString**(**args*, ***kw*)

Encodes a string into a signed string, and base64 encodes both the signature string and a random nonce.

It is up to you to provide a secret, and to keep the secret handy and consistent.

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

badsig: Signature is not correct

empty: Please enter a value

malformed: Value does not contain a signature

noneType: The input must be a string (not None)

class formencode.validators.**FieldStorageUploadConverter**(**args*, ***kw*)

Handles cgi.FieldStorage instances that are file uploads.

This doesn't do any conversion, but it can detect empty upload fields (which appear like normal fields, but have no filename when no upload was given).

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

noneType: The input must be a string (not None)

class formencode.validators.**FileUploadKeeper**(**args*, ***kw*)

Takes two inputs (a dictionary with keys `static` and `upload`) and converts them into one value on the Python side (a dictionary with `filename` and `content` keys). The upload takes priority over the static value. The filename may be None if it can't be discovered.

Handles uploads of both text and `cgi.FieldStorage` upload values.

This is basically for use when you have an upload field, and you want to keep the upload around even if the rest of the form submission fails. When converting *back* to the form submission, there may be extra values `'original_filename'` and `'original_content'`, which may want to use in your form to show the user you still have their content around.

To use this, make sure you are using variabledecode, then use something like:

```
<input type="file" name="myfield.upload">
<input type="hidden" name="myfield.static">
```

Then in your scheme:

```
class MyScheme(Scheme):
    myfield = FileUploadKeeper()
```

Note that big file uploads mean big hidden fields, and lots of bytes passed back and forth in the case of an error.

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

noneType: The input must be a string (not None)

URLs, Email, etc.

class formencode.validators.**Email**(**args*, ***kw*)

Validate an email address.

If you pass `resolve_domain=True`, then it will try to resolve the domain name to make sure it's valid. This takes longer, of course. You must have the dnspython modules installed to look up DNS (MX and A) records.

```
>>> e = Email()
>>> e.to_python(' test@foo.com ')
'test@foo.com'
>>> e.to_python('test')
Traceback (most recent call last):
    ...
Invalid: An email address must contain a single @
>>> e.to_python('test@foobar')
Traceback (most recent call last):
    ...
Invalid: The domain portion of the email address is invalid (the portion after␣
↪the @: foobar)
>>> e.to_python('test@foobar.com.5')
Traceback (most recent call last):
    ...
Invalid: The domain portion of the email address is invalid (the portion after␣
↪the @: foobar.com.5)
>>> e.to_python('test@foo..bar.com')
Traceback (most recent call last):
    ...
Invalid: The domain portion of the email address is invalid (the portion after␣
↪the @: foo..bar.com)
>>> e.to_python('test@.foo.bar.com')
Traceback (most recent call last):
    ...
```

(continues on next page)

(continued from previous page)

```
Invalid: The domain portion of the email address is invalid (the portion after␣
↪the @: .foo.bar.com)
>>> e.to_python('nobody@xn--m7r7ml7t24h.com')
'nobody@xn--m7r7ml7t24h.com'
>>> e.to_python('o*reilly@test.com')
'o*reilly@test.com'
>>> e = Email(resolve_domain=True)
>>> e.resolve_domain
True
>>> e.to_python('doesnotexist@colorstudy.com')
'doesnotexist@colorstudy.com'
>>> e.to_python('test@nyu.edu')
'test@nyu.edu'
>>> # NOTE: If you do not have dnspython installed this example won't work:
>>> e.to_python('test@thisdomaindoesnotexistithinkforsure.com')
Traceback (most recent call last):
    ...
Invalid: The domain of the email address does not exist (the portion after the @:␣
↪thisdomaindoesnotexistithinkforsure.com)
>>> e.to_python('test@google.com')
'test@google.com'
>>> e = Email(not_empty=False)
>>> e.to_python('')
```

Messages

badDomain: The domain portion of the email address is invalid (the portion after the @: `%(domain)s`)

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

badUsername: The username portion of the email address is invalid (the portion before the @: `%(username)s`)

domainDoesNotExist: The domain of the email address does not exist (the portion after the @: `%(domain)s`)

empty: Please enter an email address

noAt: An email address must contain a single @

noneType: The input must be a string (not None)

socketError: An error occured when trying to connect to the server: `%(error)s`

class formencode.validators.**URL**(**args*, ***kw*)

Validate a URL, either http://... or https://. If check_exists is true, then we'll actually make a request for the page.

If add_http is true, then if no scheme is present we'll add http://

```
>>> u = URL(add_http=True)
>>> u.to_python('foo.com')
'http://foo.com'
>>> u.to_python('http://hahaha.ha/bar.html')
'http://hahaha.ha/bar.html'
>>> u.to_python('http://xn--m7r7ml7t24h.com')
'http://xn--m7r7ml7t24h.com'
>>> u.to_python('http://xn--c1aay4a.xn--p1ai')
'http://xn--c1aay4a.xn--p1ai'
>>> u.to_python('http://foo.com/test?bar=baz&fleem=morx')
```

(continues on next page)

(continued from previous page)

```
'http://foo.com/test?bar=baz&fleem=morx'
>>> u.to_python('http://foo.com/login?came_from=http%3A%2F%2Ffoo.com%2Ftest')
'http://foo.com/login?came_from=http%3A%2F%2Ffoo.com%2Ftest'
>>> u.to_python('http://foo.com:8000/test.html')
'http://foo.com:8000/test.html'
>>> u.to_python('http://foo.com/something\\nelse')
Traceback (most recent call last):
    ...
Invalid: That is not a valid URL
>>> u.to_python('https://test.com')
'https://test.com'
>>> u.to_python('http://test')
Traceback (most recent call last):
    ...
Invalid: You must provide a full domain name (like test.com)
>>> u.to_python('http://test..com')
Traceback (most recent call last):
    ...
Invalid: That is not a valid URL
>>> u = URL(add_http=False, check_exists=True)
>>> u.to_python('http://google.com')
'http://google.com'
>>> u.to_python('google.com')
Traceback (most recent call last):
    ...
Invalid: You must start your URL with http://, https://, etc
>>> u.to_python('http://www.formencode.org/does/not/exist/page.html')
Traceback (most recent call last):
    ...
Invalid: The server responded that the page could not be found
>>> u.to_python('http://this.domain.does.not.exist.example.org/test.html')
...
Traceback (most recent call last):
    ...
Invalid: An error occured when trying to connect to the server: ...
```

If you want to allow addresses without a TLD (e.g., `localhost`) you can do:

```
>>> URL(require_tld=False).to_python('http://localhost')
'http://localhost'
```

By default, internationalized domain names (IDNA) in Unicode will be accepted and encoded to ASCII using Punycode (as described in RFC 3490). You may set allow_idna to False to change this behavior:

```
>>> URL(allow_idna=True).to_python(
... u'http://\u0433\u0443\u0433\u043b.\u0440\u0444')
'http://xn--c1aay4a.xn--p1ai'
>>> URL(allow_idna=True, add_http=True).to_python(
... u'\u0433\u0443\u0433\u043b.\u0440\u0444')
'http://xn--c1aay4a.xn--p1ai'
>>> URL(allow_idna=False).to_python(
... u'http://\u0433\u0443\u0433\u043b.\u0440\u0444')
Traceback (most recent call last):
...
Invalid: That is not a valid URL
```

Messages

badType: The input must be a string (not a %(type)s: %(value)r)

badURL: That is not a valid URL

empty: Please enter a value

httpError: An error occurred when trying to access the URL: %(error)s

noScheme: You must start your URL with http://, https://, etc

noTLD: You must provide a full domain name (like %(domain)s.com)

noneType: The input must be a string (not None)

notFound: The server responded that the page could not be found

socketError: An error occured when trying to connect to the server: %(error)s

status: The server responded with a bad status code (%(status)s)

class formencode.validators.**IPAddress**(**args*, ***kw*)
Formencode validator to check whether a string is a correct IP address.

Examples:

```
>>> ip = IPAddress()
>>> ip.to_python('127.0.0.1')
'127.0.0.1'
>>> ip.to_python('299.0.0.1')
Traceback (most recent call last):
    ...
Invalid: The octets must be within the range of 0-255 (not '299')
>>> ip.to_python('192.168.0.1/1')
Traceback (most recent call last):
    ...
Invalid: Please enter a valid IP address (a.b.c.d)
>>> ip.to_python('asdf')
Traceback (most recent call last):
    ...
Invalid: Please enter a valid IP address (a.b.c.d)
```

Messages

badFormat: Please enter a valid IP address (a.b.c.d)

badType: The input must be a string (not a %(type)s: %(value)r)

empty: Please enter a value

illegalOctets: The octets must be within the range of 0-255 (not %(octet)r)

leadingZeros: The octets must not have leading zeros

noneType: The input must be a string (not None)

class formencode.validators.**CIDR**(**args*, ***kw*)
Formencode validator to check whether a string is in correct CIDR notation (IP address, or IP address plus /mask).

Examples:

```
>>> cidr = CIDR()
>>> cidr.to_python('127.0.0.1')
'127.0.0.1'
>>> cidr.to_python('299.0.0.1')
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
    ...
Invalid: The octets must be within the range of 0-255 (not '299')
>>> cidr.to_python('192.168.0.1/1')
Traceback (most recent call last):
    ...
Invalid: The network size (bits) must be within the range of 8-32 (not '1')
>>> cidr.to_python('asdf')
Traceback (most recent call last):
    ...
Invalid: Please enter a valid IP address (a.b.c.d) or IP network (a.b.c.d/e)
```

Messages

badFormat: Please enter a valid IP address (a.b.c.d) or IP network (a.b.c.d/e)

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

illegalBits: The network size (bits) must be within the range of 8-32 (not `%(bits)r`)

illegalOctets: The octets must be within the range of 0-255 (not `%(octet)r`)

leadingZeros: The octets must not have leading zeros

noneType: The input must be a string (not None)

class formencode.validators.**MACAddress**(**args*, ***kw*)

Formencode validator to check whether a string is a correct hardware (MAC) address.

Examples:

```
>>> mac = MACAddress()
>>> mac.to_python('aa:bb:cc:dd:ee:ff')
'aabbccddeeff'
>>> mac.to_python('aa:bb:cc:dd:ee:ff:e')
Traceback (most recent call last):
    ...
Invalid: A MAC address must contain 12 digits and A-F; the value you gave has 13␣
↪characters
>>> mac.to_python('aa:bb:cc:dd:ee:fx')
Traceback (most recent call last):
    ...
Invalid: MAC addresses may only contain 0-9 and A-F (and optionally :), not 'x'
>>> MACAddress(add_colons=True).to_python('aabbccddeeff')
'aa:bb:cc:dd:ee:ff'
```

Messages

badCharacter: MAC addresses may only contain 0-9 and A-F (and optionally :), not `%(char)r`

badLength: A MAC address must contain 12 digits and A-F; the value you gave has `%(length)s` characters

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

noneType: The input must be a string (not None)

Form-wide Validation

class formencode.validators.**FormValidator**(**args*, ***kw*)

A FormValidator is something that can be chained with a Schema.

Unlike normal chaining the FormValidator can validate forms that aren't entirely valid.

The important method is .validate(), of course. It gets passed a dictionary of the (processed) values from the form. If you have .validate_partial_form set to True, then it will get the incomplete values as well – check with the "in" operator if the form was able to process any particular field.

Anyway, .validate() should return a string or a dictionary. If a string, it's an error message that applies to the whole form. If not, then it should be a dictionary of fieldName: errorMessage. The special key "form" is the error message for the form as a whole (i.e., a string is equivalent to {"form": string}).

Returns None on no errors.

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

noneType: The input must be a string (not None)

class formencode.validators.**RequireIfMissing**(**args*, ***kw*)

Require one field based on another field being present or missing.

This validator is applied to a form, not an individual field (usually using a Schema's `pre_validators` or `chained_validators`) and is available under both names `RequireIfMissing` and `RequireIfPresent`.

If you provide a `missing` value (a string key name) then if that field is missing the field must be entered. This gives you an either/or situation.

If you provide a `present` value (another string key name) then if that field is present, the required field must also be present.

```
>>> from formencode import validators
>>> v = validators.RequireIfPresent('phone_type', present='phone')
>>> v.to_python(dict(phone_type='', phone='510 420  4577'))
Traceback (most recent call last):
    ...
Invalid: You must give a value for phone_type
>>> v.to_python(dict(phone=''))
{'phone': ''}
```

Note that if you have a validator on the optionally-required field, you should probably use `if_missing=None`. This way you won't get an error from the Schema about a missing value. For example:

```
class PhoneInput(Schema):
    phone = PhoneNumber()
    phone_type = String(if_missing=None)
    chained_validators = [RequireIfPresent('phone_type', present='phone')]
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

noneType: The input must be a string (not None)

class formencode.validators.**RequireIfMatching**(**args*, ***kw*)
Require a list of fields based on the value of another field.

This validator is applied to a form, not an individual field (usually using a Schema's `pre_validators` or `chained_validators`).

You provide a field name, an expected value and a list of required fields (a list of string key names). If the value of the field, if present, matches the value of `expected_value`, then the validator will raise an `Invalid` exception for every field in `required_fields` that is missing.

```
>>> from formencode import validators
>>> v = validators.RequireIfMatching('phone_type', expected_value='mobile',␣
↪required_fields=['mobile'])
>>> v.to_python(dict(phone_type='mobile'))
Traceback (most recent call last):
    ...
Invalid: You must give a value for mobile
>>> v.to_python(dict(phone_type='someothervalue'))
{'phone_type': 'someothervalue'}
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

noneType: The input must be a string (not None)

class formencode.validators.**FieldsMatch**(**args*, ***kw*)
Tests that the given fields match, i.e., are identical. Useful for password+confirmation fields. Pass the list of field names in as *field_names*.

```
>>> f = FieldsMatch('pass', 'conf')
>>> sorted(f.to_python({'pass': 'xx', 'conf': 'xx'}).items())
[('conf', 'xx'), ('pass', 'xx')]
>>> f.to_python({'pass': 'xx', 'conf': 'yy'})
Traceback (most recent call last):
    ...
Invalid: conf: Fields do not match
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

invalid: Fields do not match (should be `%(match)s`)

invalidNoMatch: Fields do not match

noneType: The input must be a string (not None)

notDict: Fields should be a dictionary

Credit Cards

class formencode.validators.**CreditCardValidator**(**args*, ***kw*)
Checks that credit card numbers are valid (if not real).

You pass in the name of the field that has the credit card type and the field with the credit card number. The credit card type should be one of "visa", "mastercard", "amex", "dinersclub", "discover", "jcb".

You must check the expiration date yourself (there is no relation between CC number/types and expiration dates).

```
>>> cc = CreditCardValidator()
>>> sorted(cc.to_python({'ccType': 'visa', 'ccNumber': '4111111111111111'}).
↪items())
[('ccNumber', '4111111111111111'), ('ccType', 'visa')]
>>> cc.to_python({'ccType': 'visa', 'ccNumber': '411111111111111'})
Traceback (most recent call last):
    ...
Invalid: ccNumber: You did not enter a valid number of digits
>>> cc.to_python({'ccType': 'visa', 'ccNumber': '411111111111112'})
Traceback (most recent call last):
    ...
Invalid: ccNumber: You did not enter a valid number of digits
>>> cc().to_python({})
Traceback (most recent call last):
    ...
Invalid: The field ccType is missing
```

Messages

badLength: You did not enter a valid number of digits

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

invalidNumber: That number is not valid

missing_key: The field `%(key)s` is missing

noneType: The input must be a string (not None)

notANumber: Please enter only the number, no other characters

class formencode.validators.**CreditCardExpires**(**args*, ***kw*)

Checks that credit card expiration date is valid relative to the current date.

You pass in the name of the field that has the credit card expiration month and the field with the credit card expiration year.

```
>>> ed = CreditCardExpires()
>>> sorted(ed.to_python({'ccExpiresMonth': '11', 'ccExpiresYear': '2250'}).
↪items())
[('ccExpiresMonth', '11'), ('ccExpiresYear', '2250')]
>>> ed.to_python({'ccExpiresMonth': '10', 'ccExpiresYear': '2005'})
Traceback (most recent call last):
    ...
Invalid: ccExpiresMonth: Invalid Expiration Date<br>
ccExpiresYear: Invalid Expiration Date
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

invalidNumber: Invalid Expiration Date

noneType: The input must be a string (not None)

notANumber: Please enter numbers only for month and year

class formencode.validators.**CreditCardSecurityCode**(**args*, ***kw*)

Checks that credit card security code has the correct number of digits for the given credit card type.

You pass in the name of the field that has the credit card type and the field with the credit card security code.

```
>>> code = CreditCardSecurityCode()
>>> sorted(code.to_python({'ccType': 'visa', 'ccCode': '111'}).items())
[('ccCode', '111'), ('ccType', 'visa')]
>>> code.to_python({'ccType': 'visa', 'ccCode': '1111'})
Traceback (most recent call last):
    ...
Invalid: ccCode: Invalid credit card security code length
```

Messages

badLength: Invalid credit card security code length

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

noneType: The input must be a string (not None)

notANumber: Please enter numbers only for credit card security code

3.7.14 formencode.variabledecode – Turn flat HTML form submissions into nested structures

Takes GET/POST variable dictionary, as might be returned by `cgi`, and turns them into lists and dictionaries.

Keys (variable names) can have subkeys, with a `.` and can be numbered with `-`, like `a.b-3=something` means that the value `a` is a dictionary with a key `b`, and `b` is a list, the third(-ish) element with the value `something`. Numbers are used to sort, missing numbers are ignored.

This doesn't deal with multiple keys, like in a query string of `id=10&id=20`, which returns something like `{'id': ['10', '20']}`. That's left to someplace else to interpret. If you want to represent lists in this model, you use indexes, and the lists are explicitly ordered.

If you want to change the character that determines when to split for a dict or list, both variable_decode and variable_encode take dict_char and list_char keyword args. For example, to have the GET/POST variables, `a_1=something` as a list, you would use a `list_char='_'`.

Module Contents

formencode.variabledecode.**variable_decode**(*d*, *dict_char='.'*, *list_char='-'*)

Decode the flat dictionary d into a nested structure.

formencode.variabledecode.**variable_encode**(*d*, *prepend=''*, *result=None*, *add_repetitions=True*, *dict_char='.'*, *list_char='-'*)

Encode a nested structure into a flat dictionary.

class formencode.variabledecode.**NestedVariables**(**args*, ***kw*)

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

noneType: The input must be a string (not None)

CHAPTER 4

Other Links

4.1 FormEncode Community

Discussion takes place on the mailing list, formencode-discuss@lists.sf.net:

- Subcribe
- Archives

Updates to the repository are emailed to the (inaccurately named) mailing list formencode-cvs@lists.sf.net:

- Subscribe

The repository can be checked out with:

```
git clone git://github.com/formencode/formencode.git
```

Note: the SourceForge CVS repository and svn.colorstudy.com Subversion repositories and bitbucket Mercurial repositories are out of date and no longer used.

Bugs can be reported in the github bug tracker.

4.2 Downloads

The most current downloads are always available on the FormEncode Cheese Shop Page.

The repository can be checked out with:

```
git clone git://github.com/formencode/formencode.git
```

Or you can download the repository as a zip file from here:

```
https://github.com/formencode/formencode/zipball/master
```

CHAPTER 5

Indices and Search

- genindex
- modindex
- search

CHAPTER 6

Project Hosting

Python Module Index

f

Index

Symbols

A

B

C

D

E

F