
WTForms Documentation (3.0.x)

Release 3.0.1

WTForms

Oct 03, 2022

CONTENTS

| | | |
|----------|----------------------------|-----------|
| 1 | Help | 3 |
| 2 | API | 15 |
| 3 | Information | 51 |
| | Python Module Index | 65 |
| | Index | 67 |

WTForms is a flexible forms validation and rendering library for Python web development. It can work with whatever web framework and template engine you choose. It supports data validation, CSRF protection, internationalization (I18N), and more. There are various community libraries that provide closer integration with popular frameworks.

1.1 FAQ

This contains the most commonly asked questions about WTForms. The most current version of this document can always be found on the [WTForms Website](#).

1.1.1 Does WTForms work with [library here]?

The answer is most likely **yes**. WTForms tries to provide as usable an API as possible. We've listed here some of the known libraries to work with WTForms, but if it's not listed, it doesn't mean it won't work.

- **Request/Form Input**

- Django
- Webob (Includes Pylons, Google App Engine, Turbogears)
- Werkzeug (Includes Flask, Tipfy)
- any other cgi.FieldStorage-type multidict

- **Templating Engines**

- Jinja
- Mako
- Django Templates (To get the full power of WTForms in your templates, use [WTForms-Django](#).)
- Genshi

- **Database Objects**

- Pretty much any ORM or object-DB should work, as long as data objects allow attribute access to their members.

Special support is there for [SQLAlchemy](#), [Google App Engine](#), and [Django](#) collections via companion packages.

1.1.2 Does WTForms support unicode?

Simple answer: Yes.

Longer answer: WTForms uses unicode strings throughout the source code, and assumes that form input has already been coerced to unicode by your framework (Most frameworks already do this.) WTForms fields render to unicode strings by default, and therefore as long as your templating engine can work with that, you should have no unicode issues.

1.1.3 What versions of Python are supported?

WTForms supports Python 3.7+

1.1.4 How can I contribute to WTForms?

WTForms is not that scary. Really. We try to keep it as succinct and readable as possible. For bugs and feature requests, you can file a ticket on the [project page](#).

1.1.5 How do I mark in a template when a field is required?

Some validators (notably Required and Optional) set flags on the fields' *flags* object. To use this in a template, you can do something like:

```
{% for field in form %}
    {{ field }}
    {% if field.flags.required %}*{% endif %}{{ field.label }}
{% endfor %}
```

1.1.6 Does WTForms handle file uploads?

Currently, it does not. This is because WTForms strives to be framework-agnostic, and every web framework handles file uploads somewhat differently. WTForms has a *FileField* which will let you render a file input widget, but the rest is up to you. An example use in a django-ish framework:

```
class MyForm(Form):
    image = FileField()

def my_view(request):
    form = MyForm(request.POST)
    file_wrapper = request.FILES[form.image.name]
    # Do things with your file wrapper now
```

Using `form.image.name` is an easy way to know what input name was generated for your file input, even if the form is prefixed.

1.1.7 Why does blank input not go back to the default value?

A key design decision of WTForms was that form data -always- takes precedence when there's a form submission. That is, if a field exists on a form, and a form was posted, but that field's value was missing, it will not revert to a default, but instead store an empty value (and in some cases cause a validation error.)

This is for a number of reasons:

1. Security. If a form reverted to defaults on missing data, then an evil user could potentially cause problems by submitting a hand-coded form with key missing fields.
2. Bug-finding. If you omitted a field in your template, it might fall through to the default and you'd possibly miss it.
3. Consistency.

1.1.8 How do I... [convoluted combination of libraries]

You'll probably want to check out our *Solving Specific Problems* doc.

1.2 Solving Specific Problems

What follows is a collection of recipes that will help you tackle specific challenges that may crop up when using WTForms along with various other python frameworks.

1.2.1 Prelude: Poke it with a Stick!

The aim of WTForms is not to do it all, but rather to stick to the basics, while being compatible with as many frameworks as possible. We attempt to place useful things in the API so that developers can get what they want out of it, if the default behaviour is not desired.

For example, many fields in WTForms are iterable to allow you to access enclosed fields inside them, providing you another way to customize their rendering. Many attributes on the fields are readily available for you to use in your templates. We encourage you to use the introspection abilities of the python interpreter to find new ways to manipulate fields. When introspection fails, you should try reading the source for insight into how things work and how you can use things to your advantage.

If you come up with a solution that you feel is useful to others and wish to share it, please let us know on GitHub by raising an issue or submitting a pull request.

1.2.2 Removing Fields Per-instance

Sometimes, you create a form which has fields that aren't useful in all circumstances or to all users. While it is indeed possible with form inheritance to define a form with exactly the fields you need, sometimes it is necessary to just tweak an existing form. Luckily, forms can have fields removed post-instantiation by using the `del` keyword:

```
class MagazineIssueForm(Form):
    title = StringField()
    year = IntegerField('Year')
    month = SelectField(choices=MONTHS)

def edit_issue():
```

(continues on next page)

(continued from previous page)

```
publication = get_something_from_db()
form = MagazineIssueForm(...)

if publication.frequency == 'annual':
    del form.month

# render our form
```

Removing a field from a form will cause it to not be validated, and it will not show up when iterating the form. It's as if the field was never defined to begin with. Note that you cannot add fields in this way, as all fields must exist on the form when processing input data.

1.2.3 Dynamic Form Composition

This is a rare occurrence, but sometimes it's necessary to create or modify a form dynamically in your view. This is possible by creating internal subclasses:

```
def my_view():
    class F(MyBaseForm):
        pass

    F.username = StringField('username')
    for name in iterate_some_model_dynamically():
        setattr(F, name, StringField(name.title()))

    form = F(request.POST, ...)
    # do view stuff
```

1.2.4 Rendering Errors

In your template, you will often find yourself faced with the repetitive task of rendering errors for a form field. Here's a Jinja macro that may save you time:

```
{% macro with_errors(field) %}
<div class="form_field">
  {% if field.errors %}
    {% set css_class = 'has_error ' + kwargs.pop('class', '') %}
    {{ field(class=css_class, **kwargs) }}
    <ul class="errors">{% for error in field.errors %}<li>{{ error|e }}</li>{% _
↪endfor %}</ul>
  {% else %}
    {{ field(**kwargs) }}
  {% endif %}
</div>
{% endmacro %}
```

Usage: `{{ with_errors(form.field, style='font-weight: bold') }}`

1.2.5 Specialty Field Tricks

By using widget and field combinations, it is possible to create new behaviours and entirely new ways of displaying a form input to the user.

A classic example is easily supported using the `widget=` keyword arg, such as making a hidden field which stores and coerces integer data:

```
user_id = IntegerField(widget=HiddenInput())
```

Alternatively, you can create a field which does this by subclassing:

```
class HiddenInteger(IntegerField):
    widget = HiddenInput()
```

Some fields support even more sophisticated customization. For example, what if a multiple-select was desired where instead of using a multi-row `<select>`, a series of checkboxes was used? By using widgets, one can get that behavior very easily:

```
class MultiCheckboxField(SelectMultipleField):
    """
    A multiple-select, except displays a list of checkboxes.

    Iterating the field will produce subfields, allowing custom rendering of
    the enclosed checkbox fields.
    """
    widget = widgets.ListWidget(prefix_label=False)
    option_widget = widgets.CheckboxInput()
```

By overriding `option_widget`, our new multiple-select when iterated will now produce fields that render as checkboxes.

1.3 Crash Course

So you've cracked your knuckles and started working on that awesome python webapp you want to write. You get through writing a few pages and finally you need to tackle that loathsome task: form input handling and validation. Enter WTForms.

But why do I need *yet another* framework? Well, some webapp frameworks take the approach of associating database models with form handling. While this can be handy for very basic create/update views, chances are not every form you need can map directly to a database model. Or maybe you already use a generic form handling framework but you want to customize the HTML generation of those form fields, and define your own validation.

With WTForms, your form field HTML can be generated for you, but we let you customize it in your templates. This allows you to maintain separation of code and presentation, and keep those messy parameters out of your python code. Because we strive for loose coupling, you should be able to do that in any templating engine you like, as well.

1.3.1 Download / Installation

WTForms is available through [PyPI](#). Install it using pip:

```
pip install WTForms
```

1.3.2 Key Concepts

- *Forms* are the core container of WTForms. Forms represent a collection of fields, which can be accessed on the form dictionary-style or attribute style.
- *Fields* do most of the heavy lifting. Each field represents a *data type* and the field handles coercing form input to that datatype. For example, *IntegerField* and *StringField* represent two different data types. Fields contain a number of useful properties, such as a label, description, and a list of validation errors, in addition to the data the field contains.
- Every field has a *Widget* instance. The widget's job is rendering an HTML representation of that field. Widget instances can be specified for each field but every field has one by default which makes sense. Some fields are simply conveniences, for example *TextAreaField* is simply a *StringField* with the default widget being a *TextArea*.
- In order to specify validation rules, fields contain a list of *Validators*.

1.3.3 Getting Started

Let's get right down to business and define our first form:

```
from wtforms import Form, BooleanField, StringField, validators

class RegistrationForm(Form):
    username      = StringField('Username', [validators.Length(min=4, max=25)])
    email         = StringField('Email Address', [validators.Length(min=6, max=35)])
    accept_rules  = BooleanField('I accept the site rules', [validators.InputRequired()])
```

When you create a form, you define the fields in a way that is similar to the way many ORM's have you define their columns: By defining class variables which are instantiations of the fields.

Because forms are regular Python classes, you can easily extend them as you would expect:

```
class ProfileForm(Form):
    birthday = DateTimeField('Your Birthday', format='%m/%d/%y')
    signature = TextAreaField('Forum Signature')

class AdminProfileForm(ProfileForm):
    username = StringField('Username', [validators.Length(max=40)])
    level    = IntegerField('User Level', [validators.NumberRange(min=0, max=10)])
```

Via subclassing, *AdminProfileForm* gains all the fields already defined in *ProfileForm*. This allows you to easily share common subsets of fields between forms, such as the example above, where we are adding admin-only fields to *ProfileForm*.

Using Forms

Using a form is as simple as instantiating it. Consider the following django-like view, using the *RegistrationForm* we defined earlier:

```
def register(request):
    form = RegistrationForm(request.POST)
    if request.method == 'POST' and form.validate():
        user = User()
        user.username = form.username.data
        user.email = form.email.data
        user.save()
        redirect('register')
    return render_response('register.html', form=form)
```

First, we instantiate the form, providing it with any data available in `request.POST`. We then check if the request is made using POST, and if it is, we validate the form, and check that the user accepted the rules. If successful, we create a new `User` and assign the data from the validated form to it, and save it.

Editing existing objects

Our earlier registration example showed how to accept input and validate it for new entries, but what if we want to edit an existing object? Easy:

```
def edit_profile(request):
    user = request.current_user
    form = ProfileForm(request.POST, user)
    if request.method == 'POST' and form.validate():
        form.populate_obj(user)
        user.save()
        redirect('edit_profile')
    return render_response('edit_profile.html', form=form)
```

Here, we instantiate the form by providing both `request.POST` and the `user` object to the form. By doing this, the form will get any data that isn't present in the post data from the *user* object.

We're also using the form's *populate_obj* method to re-populate the `user` object with the contents of the validated form. This method is provided for convenience, for use when the field names match the names on the object you're providing with data. Typically, you will want to assign the values manually, but for this simple case it's perfect. It can also be useful for CRUD and admin forms.

Exploring in the console

WTForms forms are very simple container objects, and perhaps the easiest way to find out what's available to you in a form is to play around with a form in the console:

```
>>> from wtforms import Form, StringField, validators
>>> class UsernameForm(Form):
...     username = StringField('Username', [validators.Length(min=5)], default='test')
...
>>> form = UsernameForm()
>>> form['username']
```

(continues on next page)

(continued from previous page)

```
<wtforms.fields.core.StringField object at ...>
>>> form.username.data
'test'
>>> form.validate()
False
>>> form.errors
{'username': ['Field must be at least 5 characters long.']}
```

What we've found here is that when you instantiate a form, it contains instances of all the fields, which can be accessed via either dictionary-style or attribute-style. These fields have their own properties, as does the enclosing form.

When we validate the form, it returns `False`, meaning at least one validator was not satisfied. `form.errors` will give you a summary of all the errors.

```
>>> form2 = UsernameForm(username='Robert')
>>> form2.data
{'username': 'Robert'}
>>> form2.validate()
True
```

This time, we passed a new value for `username` when instantiating `UserForm`, and it was sufficient to validate the form.

How Forms get data

In addition to providing data using the first two arguments (*formdata* and *obj*), you can pass keyword arguments to populate the form. Note though that a few names are reserved: *formdata*, *obj*, and *prefix*.

formdata takes precedence over *obj*, which itself takes precedence over keyword arguments. For example:

```
def change_username(request):
    user = request.current_user
    form = ChangeUsernameForm(request.POST, user, username='silly')
    if request.method == 'POST' and form.validate():
        user.username = form.username.data
        user.save()
        return redirect('change_username')
    return render_response('change_username.html', form=form)
```

While you almost never use all three methods together in practice, it illustrates how WTForms looks up the *username* field:

1. If a form was submitted (`request.POST` is not empty), process the form input. Even if there was no form input for this field in particular, if there exists form input of any sort, then we will process the form input.
2. If there was no form input, then try the following in order:
 1. Check if *user* has an attribute named *username*.
 2. Check if a keyword argument named *username* was provided.
 3. Finally, if everything else fails, use the default value provided by the field, if any.

Validators

Validation in WTForms is done by providing a field with a set of validators to run when the containing form is validated. You provide these via the field constructor's second argument, *validators*:

```
class ChangeEmailForm(Form):
    email = StringField('Email', [validators.Length(min=6, max=120), validators.Email()])
```

You can provide any number of validators to a field. Typically, you will want to provide a custom error message:

```
class ChangeEmailForm(Form):
    email = StringField('Email', [
        validators.Length(min=6, message=_('Little short for an email address?')),
        validators.Email(message=_('That\'s not a valid email address.'))
    ])
```

It is generally preferable to provide your own messages, as the default messages by necessity are generic. This is also the way to provide localised error messages.

For a list of all the built-in validators, check the [Validators Reference](#)

Rendering Fields

Rendering a field is as simple as coercing it to a string:

```
>>> from wtforms import Form, StringField
>>> class SimpleForm(Form):
...     content = StringField('content')
...
>>> form = SimpleForm(content='foobar')
>>> str(form.content)
Markup('<input id="content" name="content" type="text" value="foobar">')
```

However, the real power comes from rendering the field with its `__call__()` method. By calling the field, you can provide keyword arguments, which will be injected as html attributes in the output:

```
>>> form.content(style="width: 200px;", class_="bar")
Markup('<input class="bar" id="content" name="content" style="width: 200px;" type="text"
↪value="foobar">')
```

Now let's apply this power to rendering a form in a [Jinja](#) template. First, our form:

```
class LoginForm(Form):
    username = StringField('Username')
    password = PasswordField('Password')

form = LoginForm()
```

And the template:

```
<form method="POST" action="/login">
  <div>{{ form.username.label }}: {{ form.username(class="css_class") }}</div>
  <div>{{ form.password.label }}: {{ form.password() }}</div>
</form>
```

Alternately, if you're using Django templates, you can use the *form_field* templatetag we provide in our Django extension, when you want to pass keyword arguments:

```
{% load wtforms %}
<form method="POST" action="/login">
  <div>
    {{ form.username.label }}:
    {% form_field form.username class="css_class" %}
  </div>
  <div>
    {{ form.password.label }}:
    {{ form.password }}
  </div>
</form>
```

Both of these will output:

```
<form method="POST" action="/login">
  <div>
    <label for="username">Username</label>:
    <input class="css_class" id="username" name="username" type="text" value="" />
  </div>
  <div>
    <label for="password">Password</label>:
    <input id="password" name="password" type="password" value="" />
  </div>
</form>
```

WTForms is template engine agnostic, and will work with anything that allows attribute access, string coercion, and/or function calls. The *form_field* templatetag is provided as a convenience as you can't pass arguments in Django templates.

Displaying Errors

Now that we have a template for our form, let's add error messages:

```
<form method="POST" action="/login">
  <div>{{ form.username.label }}: {{ form.username(class="css_class") }}</div>
  {% if form.username.errors %}
    <ul class="errors">{% for error in form.username.errors %}<li>{{ error }}</li>{%
→% endfor %}</ul>
    {% endif %}

  <div>{{ form.password.label }}: {{ form.password() }}</div>
  {% if form.password.errors %}
    <ul class="errors">{% for error in form.password.errors %}<li>{{ error }}</li>{%
→% endfor %}</ul>
    {% endif %}
</form>
```

If you prefer one big list of errors at the top, this is also easy:


```
{% if form.errors %}
  <ul class="errors">
    {% for field_name, field_errors in form.errors|dictsort if field_errors %}
      {% for error in field_errors %}
        <li>{{ form[field_name].label }}: {{ error }}</li>
      {% endfor %}
    {% endfor %}
  </ul>
{% endif %}
```

As error handling can become a rather verbose affair, it is preferable to use Jinja macros (or equivalent) to reduce boilerplate in your templates. (*example*)

Custom Validators

There are two ways to provide custom validators. By defining a custom validator and using it on a field:

```
from wtforms.validators import ValidationError

def is_42(form, field):
    if field.data != 42:
        raise ValidationError('Must be 42')

class FourtyTwoForm(Form):
    num = IntegerField('Number', [is_42])
```

Or by providing an in-form field-specific validator:

```
class FourtyTwoForm(Form):
    num = IntegerField('Number')

    def validate_num(form, field):
        if field.data != 42:
            raise ValidationError('Must be 42')
```

For more complex validators that take parameters, check the *Custom validators* section.

1.3.4 Next Steps

The crash course has just skimmed the surface on how you can begin using WTForms to handle form input and validation in your application. For more information, you'll want to check the following:

- The *WTForms documentation* has API documentation for the entire library.
- *Solving Specific Problems* can help you tackle specific integration issues with WTForms and other frameworks.

2.1 Forms

Forms provide the highest level API in WTForms. They contain your field definitions, delegate validation, take input, aggregate errors, and in general function as the glue holding everything together.

2.1.1 The Form class

class `wtforms.form.Form`
Declarative Form base class.

Construction

__init__ (*formdata=None, obj=None, prefix="", data=None, meta=None, **kwargs*)

Parameters

- **formdata** – Input data coming from the client, usually `request.form` or equivalent. Should provide a “multi dict” interface to get a list of values for a given key, such as what Werkzeug, Django, and WebOb provide.
- **obj** – Take existing data from attributes on this object matching form field attributes. Only used if `formdata` is not passed.
- **prefix** – If provided, all fields will have their name prefixed with the value. This is for distinguishing multiple forms on a single page. This only affects the HTML name for matching input data, not the Python name for matching existing data.
- **data** – Take existing data from keys in this dict matching form field attributes. `obj` takes precedence if it also has a matching attribute. Only used if `formdata` is not passed.
- **meta** – A dict of attributes to override on this form’s *meta* instance.
- **extra_filters** – A dict mapping field attribute names to lists of extra filter functions to run. Extra filters run after filters passed when creating the field. If the form has `filter_<fieldname>`, it is the last extra filter.
- **kwargs** – Merged with `data` to allow passing existing data as parameters. Overwrites any duplicate keys in `data`. Only used if `formdata` is not passed.

Initialize a Form. This is usually done in the context of a view/controller in your application. When a Form is constructed, the fields populate their input based on the `formdata`, `obj`, and `kwargs`.

Note Backing-store objects and `kwargs` are both expected to be provided with the values being already-coerced datatypes. WTForms does not check the types of incoming object-data or coerce them like it will

for *formdata* as it is expected this data is defaults or data from a backing store which this form represents. See the section on *using Forms* for more information.

Properties

data

A dict containing the data for each field.

Note that this is generated each time you access the property, so care should be taken when using it, as it can potentially be very expensive if you repeatedly access it. Typically used if you need to iterate all data in the form. If you just need to access the data for known fields, you should use *form.<field>.data*, not this proxy property.

errors

A dict containing a list of errors for each field. Empty if the form hasn't been validated, or there were no errors.

If present, the key `None` contains the content of `form_errors`.

form_errors

A list of form-level errors. Those are errors that does not concern a particular field, but the whole form consistency. Those errors are often set when overriding `validate()`.

meta

This is an object which contains various configuration options and also ability to customize the behavior of the form. See the *class Meta* doc for more information on what can be customized with the class `Meta` options.

Methods

validate(*extra_validators=None*)

Validate the form by calling `validate` on each field. Returns `True` if validation passes.

If the form defines a `validate_<fieldname>` method, it is appended as an extra validator for the field's `validate`.

Parameters `extra_validators` – A dict mapping field names to lists of extra validator methods to run. Extra validators run after validators passed when creating the field. If the form has `validate_<fieldname>`, it is the last extra validator.

populate_obj(*obj*)

Populates the attributes of the passed *obj* with data from the form's fields.

Note This is a destructive operation; Any attribute with the same name as a field will be overridden. Use with caution.

One common usage of this is an edit profile view:

```
def edit_profile(request):
    user = User.objects.get(pk=request.session['userid'])
    form = EditProfileForm(request.POST, obj=user)

    if request.POST and form.validate():
        form.populate_obj(user)
        user.save()
        return redirect('/home')
    return render_to_response('edit_profile.html', form=form)
```

In the above example, because the form isn't directly tied to the user object, you don't have to worry about any dirty data getting onto there until you're ready to move it over.

__iter__()

Iterate form fields in creation order.

```
{% for field in form %}
    <tr>
        <th>{{ field.label }}</th>
        <td>{{ field }}</td>
    </tr>
{% endfor %}
```

__contains__(name)Returns *True* if the named field is a member of this form.

2.1.2 Defining Forms

To define a form, one makes a subclass of *Form* and defines the fields declaratively as class attributes:

```
class MyForm(Form):
    first_name = StringField('First Name', validators=[validators.input_required()])
    last_name  = StringField('Last Name', validators=[validators.optional()])
```

Field names can be any valid python identifier, with the following restrictions:

- Field names are case-sensitive.
- Field names may not begin with “_” (underscore)
- Field names may not begin with “validate”

Form Inheritance

Forms may subclass other forms as needed. The new form will contain all fields of the parent form, as well as any new fields defined on the subclass. A field name re-used on a subclass causes the new definition to obscure the original.

```
class PastebinEdit(Form):
    language = SelectField('Programming Language', choices=PASTEBIN_LANGUAGES)
    code     = TextAreaField()

class PastebinEntry(PastebinEdit):
    name = StringField('User Name')
```

In-line Validators and Filters

In order to provide custom validation for a single field without needing to write a one-time-use validator, validation can be defined inline by defining a method with the convention *validate_fieldname*:

```
class SignupForm(Form):
    age = IntegerField('Age')

    def validate_age(form, field):
        if field.data < 13:
            raise ValidationError("We're sorry, you must be 13 or older to register")
```

The same principle applies for filters with the convention *filter_fieldname*:

```
class SignupForm(Form):
    name = StringField('name')

    def filter_name(form, field):
        return field.strip()
```

Note that filters are applied after processing the default and incoming data, but before validation.

2.1.3 Using Forms

A form is most often constructed in the controller code for handling an action, with the form data wrapper from the framework passed to its constructor, and optionally an ORM object. A typical view begins something like:

```
def edit_article(request):
    article = Article.get(...)
    form = MyForm(request.POST, article)
```

A typical CRUD view has a user editing an object that needs various fields updated. The Form would have fields describing the fields to be updated and the validation rules, where the attribute names of the fields match those of the attribute names on the object. The second parameter to the Form, the *obj* parameter, is used to populate form defaults on the initial view.

Note: While we did pass an object as the data source, this object data is only used if there is no POST data. If there is any POST data at all, then the object data is ignored. This is done for security and consistency reasons.

This pattern is mostly a convenience since most application controllers don't separate GET and POST requests into separate view methods.

The constructed form can then validate any input data and generate errors if invalid. Typically, the validation pattern in the view looks like:

```
if request.POST and form.validate():
    form.populate_obj(article)
    article.save()
    return redirect('/articles')
```

Note that we have it so *validate()* is only called if there is POST data. The reason we gate the validation check this way is that when there is no POST data (such as in a typical CRUD form) we don't want to cause validation errors.

Inside the gated block, we call *populate_obj()* to copy the data onto fields on the 'article' object. We also then redirect after a successful completion. The reason we redirect after the post is a best-practice associated with the *Post/Redirect/Get* design pattern.

If there is no POST data, or the data fails to validate, then the view “falls through” to the rendering portion. The Form object can be passed into the template and its attributes can be used to render the fields and also for displaying errors:

```
return render('edit.html', form=form, article=article)
```

So there we have a full simple “edit object” page setup which illustrates a best-practice way of using WTForms. This is by no means the only way to use WTForms, but just an illustration of how the various features work.

Here is the full code for the view we just made:

```
def edit_article(request):
    article = Article.get(...)
    form = MyForm(request.POST, article)

    if request.POST and form.validate():
        form.populate_obj(article)
        article.save()
        return redirect('/articles')

    return render('edit.html', form=form, article=article)
```

2.1.4 Low-Level API

Warning: This section is provided for completeness; and is aimed at authors of complementary libraries and those looking for very special behaviors. Don't use *BaseForm* unless you know exactly *why* you are using it.

For those looking to customize how WTForms works, for libraries or special applications, it might be worth using the *BaseForm* class. *BaseForm* is the parent class of *Form*, and most of the implementation logic from *Form* is actually handled by *BaseForm*.

The major difference on the surface between *BaseForm* and *Form* is that fields are not defined declaratively on a subclass of *BaseForm*. Instead, you must pass a dict of fields to the constructor. Likewise, you cannot add fields by inheritance. In addition, *BaseForm* does not provide: sorting fields by definition order, or inline *validate_foo* validators. Because of this, for the overwhelming majority of uses we recommend you use *Form* instead of *BaseForm* in your code.

What *BaseForm* provides is a container for a collection of fields, which it will bind at instantiation, and hold in an internal dict. Dict-style access on a *BaseForm* instance will allow you to access (and modify) the enclosed fields.

class wtforms.form.BaseForm

Base Form Class. Provides core behaviour like field construction, validation, and data and error proxying.

Construction

`__init__(fields, prefix="", meta=<DefaultMeta>)`

Parameters

- **fields** – A dict or sequence of 2-tuples of partially-constructed fields.
- **prefix** – If provided, all fields will have their name prefixed with the value.
- **meta** – A meta instance which is used for configuration and customization of WTForms behaviors.

```
form = BaseForm({
    'name': StringField(),
    'customer.age': IntegerField("Customer's Age")
})
```

Because *BaseForm* does not require field names to be valid identifiers, they can be most any python string. We recommend keeping it simple to avoid incompatibility with browsers and various form input frameworks where possible.

Properties

data

see [Form.data](#)

errors

see [Form.errors](#)

Methods

process(*formdata=None, obj=None, data=None, extra_filters=None, **kwargs*)

Process default and input data with each field.

Parameters

- **formdata** – Input data coming from the client, usually `request.form` or equivalent. Should provide a “multi dict” interface to get a list of values for a given key, such as what Werkzeug, Django, and WebOb provide.
- **obj** – Take existing data from attributes on this object matching form field attributes. Only used if `formdata` is not passed.
- **data** – Take existing data from keys in this dict matching form field attributes. `obj` takes precedence if it also has a matching attribute. Only used if `formdata` is not passed.
- **extra_filters** – A dict mapping field attribute names to lists of extra filter functions to run. Extra filters run after filters passed when creating the field. If the form has `filter_<fieldname>`, it is the last extra filter.
- **kwargs** – Merged with `data` to allow passing existing data as parameters. Overwrites any duplicate keys in `data`. Only used if `formdata` is not passed.

Since `BaseForm` does not take its data at instantiation, you must call this to provide form data to the enclosed fields. Accessing the field’s data before calling `process` is not recommended.

validate(*extra_validators=None*)

Validates the form by calling `validate` on each field.

Parameters **extra_validators** – If provided, is a dict mapping field names to a sequence of callables which will be passed as extra validators to the field’s `validate` method.

Returns `True` if no errors occur.

__iter__()

Iterate form fields in creation order.

Unlike [Form](#), fields are not iterated in definition order, but rather in whatever order the dict decides to yield them.

__contains__(*name*)

Returns `True` if the named field is a member of this form.

__getitem__(*name*)

Dict-style access to this form’s fields.

__setitem__(*name, value*)

Bind a field to this form.

```
form['openid.name'] = StringField()
```

Fields can be added and replaced in this way, but this must be done **before** `process()` is called, or the fields will not have the opportunity to receive input data. Similarly, changing fields after `validate()` will have undesired effects.

`__delitem__(name)`

Remove a field from this form.

The same caveats apply as with `__setitem__()`.

2.2 Fields

Fields are responsible for rendering and data conversion. They delegate to validators for data validation.

2.2.1 Field definitions

Fields are defined as members on a form in a declarative fashion:

```
class MyForm(Form):
    name = StringField('Full Name', [validators.required(), validators.
↪length(max=10)])
    address = TextAreaField('Mailing Address', [validators.optional(), validators.
↪length(max=200)])
```

When a field is defined on a form, the construction parameters are saved until the form is instantiated. At form instantiation time, a copy of the field is made with all the parameters specified in the definition. Each instance of the field keeps its own field data and errors list.

The label and validators can be passed to the constructor as sequential arguments, while all other arguments should be passed as keyword arguments. Some fields (such as [SelectField](#)) can also take additional field-specific keyword arguments. Consult the built-in fields reference for information on those.

2.2.2 The Field base class

class `wtforms.fields.Field`

Stores and processes data, and generates HTML for a form field.

Field instances contain the data of that instance as well as the functionality to render it within your Form. They also contain a number of properties which can be used within your templates to render the field and label.

Construction

`__init__(label=None, validators=None, filters=(), description="", id=None, default=None, widget=None, render_kw=None, name=None, _form=None, _prefix="", _translations=None, _meta=None)`

Construct a new field.

Parameters

- **label** – The label of the field.
- **validators** – A sequence of validators to call when *validate* is called.
- **filters** – A sequence of filters which are run on input data by *process*.
- **description** – A description for the field, typically used for help text.
- **id** – An id to use for the field. A reasonable default is set by the form, and you shouldn't need to set this manually.
- **default** – The default value to assign to the field, if no form or object input is provided. May be a callable.

- **widget** – If provided, overrides the widget used to render the field.
- **render_kw** (*dict*) – If provided, a dictionary which provides default keywords that will be given to the widget at render time.
- **name** – The HTML name of this field. The default value is the Python attribute name.
- **_form** – The form holding this field. It is passed by the form itself during construction. You should never pass this value yourself.
- **_prefix** – The prefix to prepend to the form name of this field, passed by the enclosing form during construction.
- **_translations** – A translations object providing message translations. Usually passed by the enclosing form during construction. See [I18n docs](#) for information on message translations.
- **_meta** – If provided, this is the ‘meta’ instance from the form. You usually don’t pass this yourself.

If *_form* isn’t provided, an `UnboundField` will be returned instead. Call its `bind()` method with a form instance and a name to construct the field.

Validation

To validate the field, call its `validate` method, providing a form and any extra validators needed. To extend validation behaviour, override `pre_validate` or `post_validate`.

validate(*form*, *extra_validators*=())

Validates the field and returns True or False. `self.errors` will contain any errors raised during validation. This is usually only called by `Form.validate`.

Subfields shouldn’t override this, but rather override either `pre_validate`, `post_validate` or both, depending on needs.

Parameters

- **form** – The form the field belongs to.
- **extra_validators** – A sequence of extra validators to run.

pre_validate(*form*)

Override if you need field-level validation. Runs before any other validators.

Parameters **form** – The form the field belongs to.

post_validate(*form*, *validation_stopped*)

Override if you need to run any field-level validation tasks after normal validation. This shouldn’t be needed in most cases.

Parameters

- **form** – The form the field belongs to.
- **validation_stopped** – *True* if any validator raised `StopValidation`.

errors

If `validate` encounters any errors, they will be inserted into this list.

Data access and processing

To handle incoming data from python, override `process_data`. Similarly, to handle incoming data from the outside, override `process_formdata`.

process(*formdata*[, *data*])

Process incoming data, calling `process_data`, `process_formdata` as needed, and run filters.

If *data* is not provided, `process_data` will be called on the field's default.

Field subclasses usually won't override this, instead overriding the `process_formdata` and `process_data` methods. Only override this for special advanced processing, such as when a field encapsulates many inputs.

Parameters *extra_filters* – A sequence of extra filters to run.

process_data(*value*)

Process the Python data applied to this field and store the result.

This will be called during form construction by the form's *kwargs* or *obj* argument.

Parameters *value* – The python object containing the value to process.

process_formdata(*valuelist*)

Process data received over the wire from a form.

This will be called during form construction with data supplied through the *formdata* argument.

Parameters *valuelist* – A list of strings to process.

data

Contains the resulting (sanitized) value of calling either of the process methods. Note that it is not HTML escaped when using in templates.

raw_data

If form data is processed, is the valuelist given from the formdata wrapper. Otherwise, *raw_data* will be *None*.

object_data

This is the data passed from an object or from kwargs to the field, stored unmodified. This can be used by templates, widgets, validators as needed (for comparison, for example)

Rendering

To render a field, simply call it, providing any values the widget expects as keyword arguments. Usually the keyword arguments are used for extra HTML attributes.

__call__(***kwargs*)

Render this field as HTML, using keyword args as additional attributes.

This delegates rendering to `meta.render_field` whose default behavior is to call the field's widget, passing any keyword arguments from this call along to the widget.

In all of the WTForms HTML widgets, keyword arguments are turned to HTML attributes, though in theory a widget is free to do anything it wants with the supplied keyword arguments, and widgets don't have to even do anything related to HTML.

If one wants to pass the "class" argument which is a reserved keyword in some python-based templating languages, one can do:

```
form.field(class_="text_blob")
```

This will output (for a text field):

```
<input type="text" name="field_name" value="blah" class="text_blob" id="field_
↪name" />
```

Note: Simply coercing the field to a string will render it as if it was called with no arguments.

`__html__()`

Returns a HTML representation of the field. For more powerful rendering, see the `__call__()` method.

Many template engines use the `__html__` method when it exists on a printed object to get an ‘html-safe’ string that will not be auto-escaped. To allow for printing a bare field without calling it, all WTForms fields implement this method as well.

Message Translations

`gettext(string)`

Get a translation for the given message.

This proxies for the internal translations object.

Parameters `string` – A string to be translated.

Returns A string which is the translated output.

`ngettext(singular, plural, n)`

Get a translation for a message which can be pluralized.

Parameters

- **singular** (`str`) – The singular form of the message.
- **plural** (`str`) – The plural form of the message.
- **n** (`int`) – The number of elements this message is referring to

Properties

`name`

The HTML form name of this field. This is the name as defined in your Form prefixed with the *prefix* passed to the Form constructor.

`short_name`

The un-prefixed name of this field.

`id`

The HTML ID of this field. If unspecified, this is generated for you to be the same as the field name.

`label`

This is a [Label](#) instance which when evaluated as a string returns an HTML `<label for="id">` construct.

`default`

This is whatever you passed as the *default* to the field’s constructor, otherwise None.

`description`

A string containing the value of the description passed in the constructor to the field; this is not HTML escaped.

`errors`

A sequence containing the validation errors for this field.

`process_errors`

Errors obtained during input processing. These will be prepended to the list of errors at validation time.

`widget`

The widget used to render the field.

`type`

The type of this field, as a string. This can be used in your templates to do logic based on the type of field:

```
{% for field in form %}
  <tr>
    {% if field.type == "BooleanField" %}
      <td></td>
      <td>{{ field }} {{ field.label }}</td>
    {% else %}
      <td>{{ field.label }}</td>
      <td>{{ field }}</td>
    {% endif %}
  </tr>
{% endfor %}
```

flags

An object containing flags set either by the field itself, or by validators on the field. For example, the built-in *InputRequired* validator sets the *required* flag. An unset flag will result in *None*.

```
{% for field in form %}
  <tr>
    <th>{{ field.label }} {% if field.flags.required %}*{% endif %}</th>
    <td>{{ field }}</td>
  </tr>
{% endfor %}
```

meta

The same *meta object* instance as is available as *Form.meta*

filters

The same sequence of filters that was passed as the *filters=* to the field constructor. This is usually a sequence of callables.

2.2.3 Basic fields

Basic fields generally represent scalar data types with single values, and refer to a single input from the form.

class `wtforms.fields.BooleanField`(*default field arguments*, *false_values=None*)

Represents an `<input type="checkbox">`. Set the checked-status by using the *default*-option. Any value for *default*, e.g. *default="checked"* puts checked into the html-element and sets the data to *True*

Parameters *false_values* – If provided, a sequence of strings each of which is an exact match string of what is considered a “false” value. Defaults to the tuple (*False*, “false”, “”)

class `wtforms.fields.DateField`(*default field arguments*, *format='%Y-%m-%d'*)

Same as *DateTimeField*, except stores a *datetime.date*.

class `wtforms.fields.DateTimeField`(*default field arguments*, *format='%Y-%m-%d %H:%M:%S'*)

A text field which stores a *datetime.datetime* matching one or several formats. If *format* is a list, any input value matching any format will be accepted, and the first format in the list will be used to produce HTML values.

class `wtforms.fields.DateTimeFieldLocal`(*default field arguments*, *format='%Y-%m-%d %H:%M:%S'*)

Same as *DateTimeField*, but represents an `<input type="datetime-local">`.

class `wtforms.fields.DecimalField`(*default field arguments*, *places=2*, *rounding=None*, *use_locale=False*, *number_format=None*)

A text field which displays and coerces data of the *decimal.Decimal* type.

Parameters

- **places** – How many decimal places to quantize the value to for display on form. If None, does not quantize value.
- **rounding** – How to round the value during quantize, for example `decimal.ROUND_UP`. If unset, uses the rounding value from the current thread's context.
- **use_locale** – If True, use locale-based number formatting. Locale-based number formatting requires the 'babel' package.
- **number_format** – Optional number format for locale. If omitted, use the default decimal format for the locale.

class wtforms.fields.**DecimalRangeField**(*default field arguments*)
Represents an <input type="range">.

class wtforms.fields.**EmailField**(*default field arguments*)
Represents an <input type="email">.

class wtforms.fields.**FileField**(*default field arguments*)
Renders a file upload field.

By default, the value will be the filename sent in the form data. WTForms **does not** deal with frameworks' file handling capabilities. A WTForms extension for a framework may replace the filename value with an object representing the uploaded data.

Example usage:

```
class UploadForm(Form):
    image = FileField('Image File', [validators.regex('^[/\\]\.jpg$')])
    description = TextAreaField('Image Description')

    def validate_image(form, field):
        if field.data:
            field.data = re.sub(r'^[a-z0-9_-]', '_', field.data)

    def upload(request):
        form = UploadForm(request.POST)
        if form.image.data:
            image_data = request.FILES[form.image.name].read()
            open(os.path.join(UPLOAD_PATH, form.image.data), 'w').write(image_data)
```

class wtforms.fields.**MultipleFileField**(*default field arguments*)
A [FileField](#) that allows choosing multiple files.

class wtforms.fields.**FloatField**(*default field arguments*)
A text field, except all input is coerced to an float. Erroneous input is ignored and will not be accepted as a value.
For the majority of uses, [DecimalField](#) is preferable to FloatField, except for in cases where an IEEE float is absolutely desired over a decimal value.

class wtforms.fields.**IntegerField**(*default field arguments*)
A text field, except all input is coerced to an integer. Erroneous input is ignored and will not be accepted as a value.

class wtforms.fields.**IntegerRangeField**(*default field arguments*)
Represents an <input type="range">.

class wtforms.fields.**MonthField**(*default field arguments, format='%Y:%m'*)
Same as [DateField](#), except represents a month, stores a `datetime.date` with `day = 1`.

class wtforms.fields.**RadioField**(*default field arguments, choices=[], coerce=str*)

Like a `SelectField`, except displays a list of radio buttons.

Iterating the field will produce subfields (each containing a label as well) in order to allow custom rendering of the individual radio fields.

```
{% for subfield in form.radio %}
    <tr>
        <td>{{ subfield }}</td>
        <td>{{ subfield.label }}</td>
    </tr>
{% endfor %}
```

Simply outputting the field without iterating its subfields will result in a `` list of radio choices.

class wtforms.fields.**SelectField**(*default field arguments, choices=[], coerce=str, option_widget=None, validate_choice=True*)

Select fields take a `choices` parameter which is either:

- a list of (value, label) pairs. It can also be a list of only values, in which case the value is used as the label. The value can be of any type, but because form data is sent to the browser as strings, you will need to provide a `coerce` function that converts a string back to the expected type.
- a dictionary of {label: list} pairs defining groupings of options.
- a function taking no argument, and returning either a list or a dictionary.

Select fields with static choice values:

```
class PastebinEntry(Form):
    language = SelectField('Programming Language', choices=[('cpp', 'C++'), ('py',
↪ 'Python'), ('text', 'Plain Text')])
```

Note that the `choices` keyword is only evaluated once, so if you want to make a dynamic drop-down list, you'll want to assign the choices list to the field after instantiation. Any submitted choices which are not in the given choices list will cause validation on the field to fail. If this option cannot be applied to your problem you may wish to skip choice validation (see below).

Select fields with dynamic choice values:

```
class UserDetails(Form):
    group_id = SelectField('Group', coerce=int)

    def edit_user(request, id):
        user = User.query.get(id)
        form = UserDetails(request.POST, obj=user)
        form.group_id.choices = [(g.id, g.name) for g in Group.query.order_by('name')]
```

Note we didn't pass a `choices` to the `SelectField` constructor, but rather created the list in the view function. Also, the `coerce` keyword arg to `SelectField` says that we use `int()` to coerce form data. The default coerce is `str()`.

Skipping choice validation:

```
class DynamicSelectForm(Form):
    dynamic_select = SelectField("Choose an option", validate_choice=False)
```

Note the `validate_choice` parameter - by setting this to `False` we are telling the `SelectField` to skip the choice validation step and instead to accept any inputted choice without checking to see if it was one of the given choices.

This should only really be used in situations where you cannot use dynamic choice values as shown above - for example where the choices of a [SelectField](#) are determined dynamically by another field on the page, such as choosing a country and state/region.

Advanced functionality

SelectField and its descendants are iterable, and iterating it will produce a list of fields each representing an option. The rendering of this can be further controlled by specifying *option_widget=*.

class wtforms.fields.SearchField(*default field arguments*)

Represents an `<input type="search">`.

class wtforms.fields.SelectMultipleField(*default field arguments, choices=[], coerce=str, option_widget=None*)

No different from a normal select field, except this one can take (and validate) multiple choices. You'll need to specify the HTML *size* attribute to the select field when rendering.

The data on the SelectMultipleField is stored as a list of objects, each of which is checked and coerced from the form input. Any submitted choices which are not in the given choices list will cause validation on the field to fail.

class wtforms.fields.SubmitField(*default field arguments*)

Represents an `<input type="submit">`. This allows checking if a given submit button has been pressed.

class wtforms.fields.StringField(*default field arguments*)

This field is the base for most of the more complicated fields, and represents an `<input type="text">`.

```
{{ form.username(size=30, maxlength=50) }}
```

class wtforms.fields.TelField(*default field arguments*)

Represents an `<input type="tel">`.

class wtforms.fields.TimeField(*default field arguments, format='%H:%M'*)

Same as [DateTimeField](#), except stores a `datetime.time`.

class wtforms.fields.URLField(*default field arguments*)

Represents an `<input type="url">`.

2.2.4 Convenience Fields

class wtforms.fields.HiddenField(*default field arguments*)

HiddenField is a convenience for a StringField with a HiddenInput widget.

It will render as an `<input type="hidden">` but otherwise coerce to a string.

HiddenField is useful for providing data from a model or the application to be used on the form handler side for making choices or finding records. Very frequently, CRUD forms will use the hidden field for an object's id.

Hidden fields are like any other field in that they can take validators and values and be accessed on the form object. You should consider validating your hidden fields just as you'd validate an input field, to prevent from malicious people playing with your data.

class wtforms.fields.PasswordField(*default field arguments*)

A StringField, except renders an `<input type="password">`.

Also, whatever value is accepted by this field is not rendered back to the browser like normal fields.

class wtforms.fields.TextAreaField(*default field arguments*)

This field represents an HTML `<textarea>` and can be used to take multi-line input.

2.2.5 Field Enclosures

Field enclosures allow you to have fields which represent a collection of fields, so that a form can be composed of multiple re-usable components or more complex data structures such as lists and nested objects can be represented.

class wtforms.fields.**FormField**(*form_class*, *default field arguments*, *separator*='-')

Encapsulate a form as a field in another form.

Parameters

- **form_class** – A subclass of Form that will be encapsulated.
- **separator** – A string which will be suffixed to this field's name to create the prefix to enclosed fields. The default is fine for most uses.

FormFields are useful for editing child objects or enclosing multiple related forms on a page which are submitted and validated together. While subclassing forms captures most desired behaviours, sometimes for reusability or purpose of combining with *FieldList*, FormField makes sense.

For example, take the example of a contact form which uses a similar set of three fields to represent telephone numbers:

```
class TelephoneForm(Form):
    country_code = IntegerField('Country Code', [validators.required()])
    area_code    = IntegerField('Area Code/Exchange', [validators.required()])
    number       = StringField('Number')

class ContactForm(Form):
    first_name   = StringField()
    last_name    = StringField()
    mobile_phone = FormField(TelephoneForm)
    office_phone = FormField(TelephoneForm)
```

In the example, we reused the TelephoneForm to encapsulate the common telephone entry instead of writing a custom field to handle the 3 sub-fields. The *data* property of the mobile_phone field will return the *data* dict of the enclosed form. Similarly, the *errors* property encapsulate the forms' errors.

class wtforms.fields.**FieldList**(*unbound_field*, *default field arguments*, *min_entries*=0, *max_entries*=None, *separator*='-')

Encapsulate an ordered list of multiple instances of the same field type, keeping data as a list.

```
>>> authors = FieldList(StringField('Name', [validators.DataRequired()])))
```

Parameters

- **unbound_field** – A partially-instantiated field definition, just like that would be defined on a form directly.
- **min_entries** – if provided, always have at least this many entries on the field, creating blank ones if the provided input does not specify a sufficient amount.
- **max_entries** – accept no more than this many entries as input, even if more exist in form-data.
- **separator** – A string which will be suffixed to this field's name to create the prefix to enclosed list entries. The default is fine for most uses.

Note: Due to a limitation in how HTML sends values, FieldList cannot enclose *BooleanField* or *SubmitField* instances.

append_entry(*[data]*)

Create a new entry with optional default data.

Entries added in this way will *not* receive formdata however, and can only receive object data.

pop_entry()

Removes the last entry from the list and returns it.

entries

Each entry in a FieldList is actually an instance of the field you passed in. Iterating, checking the length of, and indexing the FieldList works as expected, and proxies to the enclosed entries list.

Do not resize the entries list directly, this will result in undefined behavior. See *append_entry* and *pop_entry* for ways you can manipulate the list.

__iter__()**__len__**()**__getitem__**(*index*)

FieldList is not limited to enclosing simple fields; and can indeed represent a list of enclosed forms by combining FieldList with FormField:

```
class IMForm(Form):
    protocol = SelectField(choices=[('aim', 'AIM'), ('msn', 'MSN')])
    username = StringField()

class ContactForm(Form):
    first_name = StringField()
    last_name = StringField()
    im_accounts = FieldList(FormField(IMForm))
```

2.2.6 Custom Fields

While WTForms provides customization for existing fields using widgets and keyword argument attributes, sometimes it is necessary to design custom fields to handle special data types in your application.

Let's design a field which represents a comma-separated list of tags:

```
class TagListField(Field):
    widget = TextInput()

    def _value(self):
        if self.data:
            return ', '.join(self.data)
        else:
            return ''

    def process_formdata(self, valuelist):
        if valuelist:
            self.data = [x.strip() for x in valuelist[0].split(',') ]
        else:
            self.data = []
```

The *_value* method is called by the *TextInput* widget to provide the value that is displayed in the form. Overriding the *process_formdata()* method processes the incoming form data back into a list of tags.

Fields With Custom Constructors

Custom fields can also override the default field constructor if needed to provide additional customization:

```
class BetterTagListField(TagListField):
    def __init__(self, label=None, validators=None, remove_duplicates=True, **kwargs):
        super(BetterTagListField, self).__init__(label, validators, **kwargs)
        self.remove_duplicates = remove_duplicates

    def process_formdata(self, valuelist):
        super(BetterTagListField, self).process_formdata(valuelist)
        if self.remove_duplicates:
            self.data = list(self._remove_duplicates(self.data))

    @classmethod
    def _remove_duplicates(cls, seq):
        """Remove duplicates in a case insensitive, but case preserving manner"""
        d = {}
        for item in seq:
            if item.lower() not in d:
                d[item.lower()] = True
            yield item
```

When you override a Field's constructor, to maintain consistent behavior, you should design your constructor so that:

- You take `label=""`, `validators=None` as the first two positional arguments
- Add any additional arguments your field takes as keyword arguments after the label and validators
- Take `**kwargs` to catch any additional keyword arguments.
- Call the Field constructor first, passing the first two positional arguments, and all the remaining keyword args.

Considerations for overriding process()

For the vast majority of fields, it is not necessary to override `Field.process()`. Most of the time, you can achieve what is needed by overriding `process_data` and/or `process_formdata`. However, for special types of fields, such as form enclosures and other special cases of handling multiple values, it may be needed.

If you are going to override `process()`, be careful about how you deal with the `formdata` parameter. For compatibility with the maximum number of frameworks, we suggest you limit yourself to manipulating `formdata` in the following ways only:

- Testing emptiness: `if formdata`
- Checking for key existence: `key in formdata`
- Iterating all keys: `for key in formdata` (note that some wrappers may return multiple instances of the same key)
- Getting the list of values for a key: `formdata.getlist(key)`.

Most importantly, you should not use dictionary-style access to work with your `formdata` wrapper, because the behavior of this is highly variant on the wrapper: some return the first item, others return the last, and some may return a list.

2.2.7 Additional Helper Classes

class wtforms.fields.Flags

Holds a set of flags as attributes.

Accessing a non-existing attribute returns None for its value.

Usage:

```
>>> flags = Flags()
>>> flags.required = True
>>> 'required' in flags
True
>>> flags.nonexistent
>>> 'nonexistent' in flags
False
```

class wtforms.fields.Label

On all fields, the *label* property is an instance of this class. Labels can be printed to yield a `<label for="field_id">Label Text</label>` HTML tag enclosure. Similar to fields, you can also call the label with additional html params.

field_id

The ID of the field which this label will reference.

text

The original label text passed to the field's constructor.

2.3 Validators

A validator simply takes an input, verifies it fulfills some criterion, such as a maximum length for a string and returns. Or, if the validation fails, raises a *ValidationError*. This system is very simple and flexible, and allows you to chain any number of validators on fields.

class wtforms.validators.ValidationError(message="", *args, **kwargs)

Raised when a validator fails to validate its input.

class wtforms.validators.StopValidation(message="", *args, **kwargs)

Causes the validation chain to stop.

If StopValidation is raised, no more validators in the validation chain are called. If raised with a message, the message will be added to the errors list.

2.3.1 Built-in validators

class wtforms.validators.DataRequired(message=None)

Checks the field's data is 'truthy' otherwise stops the validation chain.

This validator checks that the *data* attribute on the field is a 'true' value (effectively, it does `if field.data`.) Furthermore, if the data is a string type, a string containing only whitespace characters is considered false.

If the data is empty, also removes prior errors (such as processing errors) from the field.

NOTE this validator used to be called *Required* but the way it behaved (requiring coerced data, not input data) meant it functioned in a way which was not symmetric to the *Optional* validator and furthermore caused confusion

with certain fields which coerced data to ‘falsey’ values like `0`, `Decimal(0)`, `time(0)` etc. Unless a very specific reason exists, we recommend using the [InputRequired](#) instead.

Parameters `message` – Error message to raise in case of a validation error.

Sets the *required* attribute on widgets.

This also sets the *required* [flag](#) on fields it is used on. This flag causes the *required* attribute to be rendered in the tag, which prevents a request/response cycle for validation. This behavior can be overridden in the following ways:

- Specifying `required=False` when rendering in the template.
- Making a custom a widget that doesn’t set it.
- Rendering the `novalidate` attribute” on the `form` tag, or the `formnovalidate` attribute on a submit button.

The *required* flag behavior also applies to the [InputRequired](#) class.

```
class wtforms.validators.Email(message=None, granular_message=False, check_deliverability=False,
                               allow_smtputf8=True, allow_empty_local=False)
```

Validates an email address. Requires `email_validator` package to be installed. For ex: `pip install wtforms[email]`.

Parameters

- **message** – Error message to raise in case of a validation error.
- **granular_message** – Use validation failed message from `email_validator` library (Default `False`).
- **check_deliverability** – Perform domain name resolution check (Default `False`).
- **allow_smtputf8** – Fail validation for addresses that would require SMTPUTF8 (Default `True`).
- **allow_empty_local** – Allow an empty local part (i.e. `@example.com`), e.g. for validating Postfix aliases (Default `False`).

```
class wtforms.validators.EqualTo(fieldname, message=None)
```

Compares the values of two fields.

Parameters

- **fieldname** – The name of the other field to compare to.
- **message** – Error message to raise in case of a validation error. Can be interpolated with `%(other_label)s` and `%(other_name)s` to provide a more helpful error.

This validator can be used to facilitate in one of the most common scenarios, the password change form:

```
class ChangePassword(Form):
    password = PasswordField('New Password', [InputRequired(), EqualTo('confirm',
↵message='Passwords must match'))
    confirm = PasswordField('Repeat Password')
```

In the example, we use the `InputRequired` validator to prevent the `EqualTo` validator from trying to see if the passwords do not match if there was no passwords specified at all. Because `InputRequired` stops the validation chain, `EqualTo` is not run in the case the password field is left empty.

```
class wtforms.validators.InputRequired(message=None)
```

Validates that input was provided for this field.

Note there is a distinction between this and `DataRequired` in that `InputRequired` looks that form-input data was provided, and `DataRequired` looks at the post-coercion data.

Sets the *required* attribute on widgets.

This also sets the required *flag* on fields it is used on. See [DataRequired](#) for a description of behavior regarding this flag.

class wtforms.validators.**IPAddress**(*ipv4=True, ipv6=False, message=None*)
Validates an IP address.

Parameters

- **ipv4** – If True, accept IPv4 addresses as valid (default True)
- **ipv6** – If True, accept IPv6 addresses as valid (default False)
- **message** – Error message to raise in case of a validation error.

class wtforms.validators.**Length**(*min=- 1, max=- 1, message=None*)
Validates the length of a string.

Parameters

- **min** – The minimum required length of the string. If not provided, minimum length will not be checked.
- **max** – The maximum length of the string. If not provided, maximum length will not be checked.
- **message** – Error message to raise in case of a validation error. Can be interpolated using *%(min)d* and *%(max)d* if desired. Useful defaults are provided depending on the existence of min and max.

When supported, sets the *minlength* and *maxlength* attributes on widgets.

class wtforms.validators.**MacAddress**(*message=None*)
Validates a MAC address.

Parameters **message** – Error message to raise in case of a validation error.

class wtforms.validators.**NumberRange**(*min=None, max=None, message=None*)
Validates that a number is of a minimum and/or maximum value, inclusive. This will work with any comparable number type, such as floats and decimals, not just integers.

Parameters

- **min** – The minimum required value of the number. If not provided, minimum value will not be checked.
- **max** – The maximum value of the number. If not provided, maximum value will not be checked.
- **message** – Error message to raise in case of a validation error. Can be interpolated using *%(min)s* and *%(max)s* if desired. Useful defaults are provided depending on the existence of min and max.

When supported, sets the *min* and *max* attributes on widgets.

class wtforms.validators.**Optional**(*strip_whitespace=True*)
Allows empty input and stops the validation chain from continuing.

If input is empty, also removes prior errors (such as processing errors) from the field.

Parameters **strip_whitespace** – If True (the default) also stop the validation chain on input which consists of only whitespace.

Sets the *optional* attribute on widgets.

This also sets the optional *flag* on fields it is used on.

class wtforms.validators.**Regex**(*regex, flags=0, message=None*)

Validates the field against a user provided regexp.

Parameters

- **regex** – The regular expression string to use. Can also be a compiled regular expression pattern.
- **flags** – The regexp flags to use, for example re.IGNORECASE. Ignored if *regex* is not a string.
- **message** – Error message to raise in case of a validation error.

class wtforms.validators.**URL**(*require_tld=True, message=None*)

Simple regexp based url validation. Much like the email validator, you probably want to validate the url later by other means if the url must resolve.

Parameters

- **require_tld** – If true, then the domain-name portion of the URL must contain a .tld suffix. Set this to false if you want to allow domains like *localhost*.
- **message** – Error message to raise in case of a validation error.

class wtforms.validators.**UUID**(*message=None*)

Validates a UUID.

Parameters **message** – Error message to raise in case of a validation error.

class wtforms.validators.**AnyOf**(*values, message=None, values_formatter=None*)

Compares the incoming data to a sequence of valid inputs.

Parameters

- **values** – A sequence of valid inputs.
- **message** – Error message to raise in case of a validation error. *%(values)s* contains the list of values.
- **values_formatter** – Function used to format the list of values in the error message.

class wtforms.validators.**NoneOf**(*values, message=None, values_formatter=None*)

Compares the incoming data to a sequence of invalid inputs.

Parameters

- **values** – A sequence of invalid inputs.
- **message** – Error message to raise in case of a validation error. *%(values)s* contains the list of values.
- **values_formatter** – Function used to format the list of values in the error message.

2.3.2 Custom validators

We will step through the evolution of writing a length-checking validator similar to the built-in `Length` validator, starting from a case-specific one to a generic reusable validator.

Let's start with a simple form with a name field and its validation:

```
class MyForm(Form):
    name = StringField('Name', [InputRequired()])

    def validate_name(form, field):
        if len(field.data) > 50:
            raise ValidationError('Name must be less than 50 characters')
```

Above, we show the use of an *in-line validator* to do validation of a single field. In-line validators are good for validating special cases, but are not easily reusable. If, in the example above, the `name` field were to be split into two fields for first name and surname, you would have to duplicate your work to check two lengths.

So let's start on the process of splitting the validator out for re-use:

```
def my_length_check(form, field):
    if len(field.data) > 50:
        raise ValidationError('Field must be less than 50 characters')

class MyForm(Form):
    name = StringField('Name', [InputRequired(), my_length_check])
```

All we've done here is move the exact same code out of the class and as a function. Since a validator can be any callable which accepts the two positional arguments form and field, this is perfectly fine, but the validator is very special-cased.

Instead, we can turn our validator into a more powerful one by making it a factory which returns a callable:

```
def length(min=-1, max=-1):
    message = 'Must be between %d and %d characters long.' % (min, max)

    def _length(form, field):
        l = field.data and len(field.data) or 0
        if l < min or max != -1 and l > max:
            raise ValidationError(message)

    return _length

class MyForm(Form):
    name = StringField('Name', [InputRequired(), length(max=50)])
```

Now we have a configurable length-checking validator that handles both minimum and maximum lengths. When `length(max=50)` is passed in your validators list, it returns the enclosed `_length` function as a closure, which is used in the field's validation chain.

This is now an acceptable validator, but we recommend that for reusability, you use the pattern of allowing the error message to be customized via passing a `message=` parameter:

```
class Length(object):
    def __init__(self, min=-1, max=-1, message=None):
        self.min = min
        self.max = max
```

(continues on next page)

(continued from previous page)

```

    if not message:
        message = 'Field must be between %i and %i characters long.' % (min, max)
    self.message = message

    def __call__(self, form, field):
        l = field.data and len(field.data) or 0
        if l < self.min or self.max != -1 and l > self.max:
            raise ValidationError(self.message)

length = Length

```

In addition to allowing the error message to be customized, we've now converted the length validator to a class. This wasn't necessary, but we did this to illustrate how one would do so. Because fields will accept any callable as a validator, callable classes are just as applicable. For complex validators, or using inheritance, you may prefer this.

We aliased the Length class back to the original length name in the above example. This allows you to keep API compatibility as you move your validators from factories to classes, and thus we recommend this for those writing validators they will share.

Setting flags on the field with validators

Sometimes, it's useful to know if a validator is present on a given field, like for use in template code. To do this, validators are allowed to specify flags which will then be available on the *field's flags object*. Some of the built-in validators such as Required already do this.

To specify flags on your validator, set the `field_flags` attribute on your validator. When the Field is constructed, the flags with the same name will be set on your field. For example, let's imagine a validator that validates that input is valid BBCode. We can set a flag on the field then to signify that the field accepts BBCode:

```

# class implementation
class ValidBBCode(object):
    def __init__(self):
        self.field_flags = {'accepts_bbcode': True}

# factory implementation
def valid_bbcode():
    def _valid_bbcode(form, field):
        pass # validator implementation here

    _valid_bbcode.field_flags = {'accepts_bbcode': True}
    return _valid_bbcode

```

Then we can check it in our template, so we can then place a note to the user:

```

{{ field(rows=7, cols=70) }}
{% if field.flags.accepts_bbcode %}
    <div class="note">This field accepts BBCode formatting as input.</div>
{% endif %}

```

Some considerations on using flags:

- Boolean flags will set HTML valueless attributes (e.g. `{required: True}` will give `<input type="text" required>`). Other flag types will set regular HTML attributes (e.g. `{maxlength: 8}` will give `<input type="text" maxlength="8">`).

- If multiple validators set the same flag, the flag will have the value set by the first validator.
- Flags are set from validators only in `Field.__init__()`, so inline validators and extra passed-in validators cannot set them.

2.4 Widgets

Widgets are classes whose purpose are to render a field to its usable representation, usually XHTML. When a field is called, the default behaviour is to delegate the rendering to its widget. This abstraction is provided so that widgets can easily be created to customize the rendering of existing fields.

Note All built-in widgets will return upon rendering a “HTML-safe” unicode string subclass that many templating frameworks (Jinja, Mako, Genshi) will recognize as not needing to be auto-escaped.

2.4.1 Built-in widgets

class `wtforms.widgets.ColorInput(input_type=None)`

Render an input with type “color”.

class `wtforms.widgets.CheckboxInput(input_type=None)`

Render a checkbox.

The checked HTML attribute is set if the field’s data is a non-false value.

class `wtforms.widgets.DateTimeInput(input_type=None)`

Render an input with type “datetime”.

class `wtforms.widgets.DateTimeLocalInput(input_type=None)`

Render an input with type “datetime-local”.

class `wtforms.widgets.DateInput(input_type=None)`

Render an input with type “date”.

class `wtforms.widgets.EmailInput(input_type=None)`

Render an input with type “email”.

class `wtforms.widgets.FileInput(multiple=False)`

Render a file chooser input.

Parameters `multiple` – allow choosing multiple files

class `wtforms.widgets.HiddenInput(*args, **kwargs)`

Render a hidden input.

class `wtforms.widgets.Input(input_type=None)`

Render a basic `<input>` field.

This is used as the basis for most of the other input fields.

By default, the `_value()` method will be called upon the associated field to provide the `value=` HTML attribute.

class `wtforms.widgets.ListWidget(html_tag='ul', prefix_label=True)`

Render a list of fields as a *ul* or *ol* list.

This is used for fields which encapsulate many inner fields as subfields. The widget will try to iterate the field to get access to the subfields and call them to render them.

If `prefix_label` is set, the subfield’s label is printed before the field, otherwise afterwards. The latter is useful for iterating radios or checkboxes.

class wtforms.widgets.**MonthInput**(*input_type=None*)

Renders an input with type “month”.

class wtforms.widgets.**NumberInput**(*step=None, min=None, max=None*)

Renders an input with type “number”.

class wtforms.widgets.**PasswordInput**(*hide_value=True*)

Render a password input.

For security purposes, this field will not reproduce the value on a form submit by default. To have the value filled in, set *hide_value* to *False*.

class wtforms.widgets.**RadioInput**(*input_type=None*)

Render a single radio button.

This widget is most commonly used in conjunction with ListWidget or some other listing, as singular radio buttons are not very useful.

class wtforms.widgets.**RangeInput**(*step=None*)

Renders an input with type “range”.

class wtforms.widgets.**SubmitInput**(*input_type=None*)

Renders a submit button.

The field’s label is used as the text of the submit button instead of the data on the field.

class wtforms.widgets.**SearchInput**(*input_type=None*)

Renders an input with type “search”.

class wtforms.widgets.**Select**(*multiple=False*)

Renders a select field.

If *multiple* is *True*, then the *size* property should be specified on rendering to make the field useful.

The field must provide an *iter_choices()* method which the widget will call on rendering; this method must yield tuples of (*value, label, selected*). It also must provide a *has_groups()* method which tells whether choices are divided into groups, and if they do, the field must have an *iter_groups()* method that yields tuples of (*label, choices*), where *choices* is an iterable of (*value, label, selected*) tuples.

class wtforms.widgets.**TableWidget**(*with_table_tag=True*)

Renders a list of fields as a set of table rows with th/td pairs.

If *with_table_tag* is *True*, then an enclosing `<table>` is placed around the rows.

Hidden fields will not be displayed with a row, instead the field will be pushed into a subsequent table row to ensure XHTML validity. Hidden fields at the end of the field list will appear outside the table.

class wtforms.widgets.**TelInput**(*input_type=None*)

Renders an input with type “tel”.

class wtforms.widgets.**TextArea**

Renders a multi-line text area.

rows and *cols* ought to be passed as keyword args when rendering.

class wtforms.widgets.**TextInput**(*input_type=None*)

Render a single-line text input.

class wtforms.widgets.**TimeInput**(*input_type=None*)

Renders an input with type “time”.

class wtforms.widgets.**URLInput**(*input_type=None*)

Renders an input with type “url”.

class wtforms.widgets.**WeekInput**(input_type=None)
 Renders an input with type “week”.

2.4.2 Widget-Building Utilities

These utilities are used in WTForms widgets to help render HTML and also in order to work along with HTML templating frameworks. They can be imported for use in building custom widgets as well.

wtforms.widgets.html_params(**kwargs)
 Generate HTML attribute syntax from inputted keyword arguments.

The output value is sorted by the passed keys, to provide consistent output each time this function is called with the same parameters. Because of the frequent use of the normally reserved keywords *class* and *for*, suffixing these with an underscore will allow them to be used.

In order to facilitate the use of data- and aria- attributes, if the name of the attribute begins with data_ or aria_, then every underscore will be replaced with a hyphen in the generated attribute.

```
>>> html_params(data_attr='user.name', aria_labeledby='name')
'data-attr="user.name" aria-labeledby="name"'
```

In addition, the values **True** and **False** are special:

- attr=True generates the HTML compact output of a boolean attribute, e.g. checked=True will generate simply checked
- attr=False will be ignored and generate no output.

```
>>> html_params(name='text1', id='f', class_='text')
'class="text" id="f" name="text1"'
>>> html_params(checked=True, readonly=False, name="text1", abc="hello")
'abc="hello" checked name="text1"'
```

Changed in version 3.0: aria_ args convert underscores to hyphens like data_ args.

Changed in version 2.2: data_ args convert all underscores to hyphens, instead of only the first one.

WTForms uses [MarkupSafe](#) to escape unsafe HTML characters before rendering. You can mark a string using `markupsafe.Markup` to indicate that it should not be escaped.

2.4.3 Custom widgets

Widgets, much like validators, provide a simple callable contract. Widgets can take customization arguments through a constructor if needed as well. When the field is called or printed, it will call the widget with itself as the first argument and then any additional arguments passed to its caller as keywords. Passing the field is done so that one instance of a widget might be used across many field instances.

Let’s look at a widget which renders a text field with an additional class if there are errors:

```
class MyTextInput(TextInput):
    def __init__(self, error_class='has_errors'):
        super(MyTextInput, self).__init__()
        self.error_class = error_class

    def __call__(self, field, **kwargs):
```

(continues on next page)

(continued from previous page)

```

if field.errors:
    c = kwargs.pop('class', '') or kwargs.pop('class_', '')
    kwargs['class'] = '%s %s' % (self.error_class, c)
return super(MyTextInput, self).__call__(field, **kwargs)

```

In the above example, we extended the behavior of the existing `TextInput` widget to append a CSS class as needed. However, widgets need not extend from an existing widget, and indeed don't even have to be a class. For example, here is a widget that renders a `SelectMultipleField` as a collection of checkboxes:

```

def select_multi_checkbox(field, ul_class='', **kwargs):
    kwargs.setdefault('type', 'checkbox')
    field_id = kwargs.pop('id', field.id)
    html = ['<ul %s>' % html_params(id=field_id, class_=ul_class)]
    for value, label, checked in field.iter_choices():
        choice_id = '%s-%s' % (field_id, value)
        options = dict(kwargs, name=field.name, value=value, id=choice_id)
        if checked:
            options['checked'] = 'checked'
        html.append('<li><input %s /> ' % html_params(**options))
        html.append('<label for="%s">%s</label></li>' % (choice_id, label))
    html.append('</ul>')
    return ''.join(html)

class TestForm(Form):
    tester = SelectMultipleField(choices=my_choices, widget=select_multi_checkbox)

```

2.5 class Meta

the *class Meta* paradigm allows WTForms features to be customized, and even new behaviors to be introduced. It also supplies a place where configuration for any complementary modules can be done.

Typical usage looks something like:

```

class MyForm(Form):
    class Meta:
        csrf = True
        locales = ('en_US', 'en')

    name = StringField(...)
    # and so on...

```

For the majority of users, using a class *Meta* is mostly going to be done for customizing options used by the default behaviors, however for completeness the entire API of the *Meta* interface is shown here.

class wtforms.meta.DefaultMeta

This is the default *Meta* class which defines all the default values and therefore also the 'API' of the class *Meta* interface.

Configuration

csrf = False

Setting `csrf` to `True` will enable CSRF for the form. The value can also be overridden per-instance via instantiation-time customization (for example, if `csrf` needs to be turned off only in a special case)

```
form = MyForm(request.form, meta={'csrf': False})
```

csrf_class = None

If set, this is a class which is used to implement CSRF protection. Read the [CSRF Documentation](#) to get more information on how to use.

csrf_field_name = 'csrf_token'

The name of the automatically added CSRF token field.

locales = False

Setting to a sequence of strings specifies the priority order of locales to try to find translations for built-in messages of WTForms.

If the value is *False*, then strings are not translated (the translations provider is replaced with a dummy provider)

example:

```
locales = ('fr_FR', 'fr')
```

Also see [Internationalization \(i18n\)](#) for more information.

cache_translations = True

If *True* (the default) then cache translation objects. The default cache is done at class-level so it's shared with all class Meta.

Advanced Customization

Usually, you do not need to override these methods, as they provide core behaviors of WTForms.

build_csrf(form)

Build a CSRF implementation. This is called once per form instance.

The default implementation builds the class referenced to by [csrf_class](#) with zero arguments. If [csrf_class](#) is *None*, will instead use the default implementation [wtforms.csrf.session.SessionCSRF](#).

Parameters **form** – The form.

Returns A CSRF implementation.

get_translations(form)

Override in subclasses to provide alternate translations factory. See the [i18n](#) documentation for more.

Parameters **form** – The form.

Returns An object that provides `gettext()` and `ngettext()` methods.

bind_field(form, unbound_field, options)

`bind_field` allows potential customization of how fields are bound.

The default implementation simply passes the options to `UnboundField.bind()`.

Parameters

- **form** – The form.
- **unbound_field** – The unbound field.
- **options** – A dictionary of options which are typically passed to the field.

Returns A bound field

wrap_formdata(form, formdata)

`wrap_formdata` allows doing custom wrappers of WTForms formdata.

The default implementation detects webob-style multidicts and wraps them, otherwise passes formdata back un-changed.

Parameters

- **form** – The form.
- **formdata** – Form data.

Returns A form-input wrapper compatible with WTForms.

render_field(*field*, *render_kw*)

render_field allows customization of how widget rendering is done.

The default implementation calls `field.widget(field, **render_kw)`

2.6 CSRF Protection

The CSRF package includes tools that help you implement checking against cross-site request forgery (“csrf”). Due to the large number of variations on approaches people take to CSRF (and the fact that many make compromises) the base implementation allows you to plug in a number of CSRF validation approaches.

CSRF implementations are made by subclassing [CSRF](#). For utility, we have provided one possible CSRF implementation in the package that can be used with many frameworks for session-based hash secure keying, [SessionCSRF](#).

2.6.1 Using CSRF

CSRF in WTForms 2.0 is now driven through a number of variables on `class Meta`. After choosing a CSRF implementation, import it and configure it on the class Meta of a subclass of Form like such:

```
from somemodule import SomeCSRF

class MyBaseForm(Form):
    class Meta:
        csrf = True # Enable CSRF
        csrf_class = SomeCSRF # Set the CSRF implementation
        csrf_secret = b'foobar' # Some implementations need a secret key.
        # Any other CSRF settings here.
```

And once you’ve got this set up, you can define your forms as a subclass of *MyBaseForm*:

```
class UserForm(MyBaseForm):
    name = StringField()
    age = IntegerField()

def view():
    form = UserForm(request.POST)
    if request.POST and form.validate():
        pass # Form is valid and CSRF succeeded

    return render('user.html', form=form)
```

There is a special field inside the CSRF form (called `csrf_token` by default) which you need to make sure you render in your template:

```
<form action="/user" method="POST">
{{ form.csrf_token }}
{% if form.csrf_token.errors %}
    <div class="warning">You have submitted an invalid CSRF token</div>
{% endif %}
<div>{{ form.name }} {{ form.name.label }}</div>
<div>{{ form.age }}{{ form.age.label }}</div>
```

Remember, with the *class Meta* you can always override variables in a sub-class or at the constructor for special-cases:

```
class SearchForm(MyBaseForm):
    """
    We expect search queries to come externally, thus we don't want CSRF
    even though it's set up on the base form.
    """
    class Meta:
        # This overrides the value from the base form.
        csrf = False
```

2.6.2 How WTForms CSRF works

Most CSRF implementations hinge around creating a special token, which is put in a hidden field on the form named *csrf_token*, which must be rendered in your template to be passed from the browser back to your view. There are many different methods of generating this token, but they are usually the result of a cryptographic hash function against some data which would be hard to forge.

class `wtforms.csrf.core.CSRFTokenField(*args, **kwargs)`

A subclass of `HiddenField` designed for sending the CSRF token that is used for most CSRF protection schemes.

Notably different from a normal field, this field always renders the current token regardless of the submitted value, and also will not be populated over to object data via `populate_obj`

__init__(*args, **kw)

Construct a new field.

Parameters

- **label** – The label of the field.
- **validators** – A sequence of validators to call when *validate* is called.
- **filters** – A sequence of filters which are run on input data by *process*.
- **description** – A description for the field, typically used for help text.
- **id** – An id to use for the field. A reasonable default is set by the form, and you shouldn't need to set this manually.
- **default** – The default value to assign to the field, if no form or object input is provided. May be a callable.
- **widget** – If provided, overrides the widget used to render the field.
- **render_kw** (*dict*) – If provided, a dictionary which provides default keywords that will be given to the widget at render time.
- **name** – The HTML name of this field. The default value is the Python attribute name.

- **_form** – The form holding this field. It is passed by the form itself during construction. You should never pass this value yourself.
- **_prefix** – The prefix to prepend to the form name of this field, passed by the enclosing form during construction.
- **_translations** – A translations object providing message translations. Usually passed by the enclosing form during construction. See [I18n docs](#) for information on message translations.
- **_meta** – If provided, this is the ‘meta’ instance from the form. You usually don’t pass this yourself.

If `_form` isn’t provided, an `UnboundField` will be returned instead. Call its `bind()` method with a form instance and a name to construct the field.

current_token = None

_value()

We want to always return the current token on render, regardless of whether a good or bad token was passed.

populate_obj(*args)

Don’t populate objects with the CSRF token

pre_validate(form)

Handle validation of this token field.

process(*args, **kwargs)

Process incoming data, calling `process_data`, `process_formdata` as needed, and run filters.

If `data` is not provided, `process_data` will be called on the field’s default.

Field subclasses usually won’t override this, instead overriding the `process_formdata` and `process_data` methods. Only override this for special advanced processing, such as when a field encapsulates many inputs.

Parameters `extra_filters` – A sequence of extra filters to run.

class `wtforms.csrf.core.CSRF`

setup_form(form)

Receive the form we’re attached to and set up fields.

The default implementation creates a single field of type `field_class` with name taken from the `csrf_field_name` of the class meta.

Parameters `form` – The form instance we’re attaching to.

Returns A sequence of `(field_name, unbound_field)` 2-tuples which are unbound fields to be added to the form.

generate_csrf_token(csrf_token_field)

Implementations must override this to provide a method with which one can get a CSRF token for this form.

A CSRF token is usually a string that is generated deterministically based on some sort of user data, though it can be anything which you can validate on a subsequent request.

Parameters `csrf_token_field` – The field which is being used for CSRF.

Returns A generated CSRF string.

validate_csrf_token(form, field)

Override this method to provide custom CSRF validation logic.

The default CSRF validation logic simply checks if the recently generated token equals the one we received as formdata.

Parameters

- **form** – The form which has this CSRF token.
- **field** – The CSRF token field.

field_class = <class 'wtforms.csrf.core.CSRFTokenField'>

The class of the token field we're going to construct. Can be overridden in subclasses if need be.

2.6.3 Creating your own CSRF implementation

Here we will sketch out a simple theoretical CSRF implementation which generates a hash token based on the user's IP.

Note This is a simplistic example meant to illustrate creating a CSRF implementation. This isn't recommended to be used in production because the token is deterministic and non-changing per-IP, which means this isn't the most secure implementation of CSRF.

First, let's create our CSRF class:

```
from wtforms.csrf.core import CSRF
from hashlib import md5

SECRET_KEY = '1234567890'

class IPAddressCSRF(CSRF):
    """
    Generate a CSRF token based on the user's IP. I am probably not very
    secure, so don't use me.
    """
    def setup_form(self, form):
        self.csrf_context = form.meta.csrf_context
        return super(IPAddressCSRF, self).setup_form(form)

    def generate_csrf_token(self, csrf_token):
        token = md5(SECRET_KEY + self.csrf_context).hexdigest()
        return token

    def validate_csrf_token(self, form, field):
        if field.data != field.current_token:
            raise ValueError('Invalid CSRF')
```

Now that we have this taken care of, let's write a simple form and view which would implement this:

```
class RegistrationForm(Form):
    class Meta:
        csrf = True
        csrf_class = IPAddressCSRF

    name = StringField('Your Name')
    email = StringField('Email', [validators.email()])

    def register(request):
```

(continues on next page)

(continued from previous page)

```

form = RegistrationForm(
    request.POST,
    meta={'csrf_context': request.ip}
)

if request.method == 'POST' and form.validate():
    pass # We're all good, create a user or whatever it is you do
elif form.csrf_token.errors:
    pass # If we're here we suspect the user of cross-site request forgery
else:
    pass # Any other errors

return render('register.html', form=form)

```

And finally, a simple template:

```

<form action="register" method="POST">
    {{ form.csrf_token }}
    <p>{{ form.name.label }}: {{ form.name }}</p>
    <p>{{ form.email.label }}: {{ form.email }}</p>
    <input type="submit" value="Register">
</form>

```

Please note that implementing CSRF detection is not fool-proof, and even with the best CSRF protection implementation, it's possible for requests to be forged by expert attackers. However, a good CSRF protection would make it infeasible for someone from an external site to hijack a form submission from another user and perform actions as them without additional a priori knowledge.

In addition, it's important to understand that very often, the more strict the CSRF protection, the higher the chance of false positives occurring (ie, legitimate users getting blocked by your CSRF protection) and choosing a CSRF implementation is actually a matter of compromise. We will attempt to provide a handful of usable reference algorithms built in to this library in the future, to allow that choice to be easy.

Some tips on criteria people often examine when evaluating CSRF implementations:

- **Reproducibility** If a token is based on attributes about the user, it gains the advantage that one does not need secondary storage in which to store the value between requests. However, if the same attributes can be reproduced by an attacker, then the attacker can potentially forge this information.
- **Reusability**. It might be desired to make a completely different token every use, and disallow users from reusing past tokens. This is an extremely powerful protection, but can have consequences on if the user uses the back button (or in some cases runs forms simultaneously in multiple browser tabs) and submits an old token, or otherwise. A possible compromise is to allow reusability in a time window (more on that later).
- **Time Ranges** Many CSRF approaches use time-based expiry to make sure that a token cannot be (re)used beyond a certain point. Care must be taken in choosing the time criteria for this to not lock out legitimate users. For example, if a user might walk away while filling out a long-ish form, or to go look for their credit card, the time for expiry should take that into consideration to provide a balance between security and limiting user inconvenience.
- **Requirements** Some CSRF-prevention methods require the use of browser cookies, and some even require client-side scripting support. The webmaster implementing the CSRF needs to consider that such requirements (though effective) may lock certain legitimate users out, and make this determination whether it is a good idea to use. For example, for a site already using cookies for login, adding another for CSRF isn't as big of a deal, but for other sites it may not be feasible.

2.6.4 Session-based CSRF implementation

A provided CSRF implementation which puts CSRF data in a session.

This can be used fairly comfortably with many *request.session* type objects, including the Werkzeug/Flask session store, Django sessions, and potentially other similar objects which use a dict-like API for storing session keys.

The basic concept is a randomly generated value is stored in the user's session, and an hmac-sha1 of it (along with an optional expiration time, for extra security) is used as the value of the *csrf_token*. If this token validates with the hmac of the random value + expiration time, and the expiration time is not passed, the CSRF validation will pass.

class wtforms.csrf.session.SessionCSRF

Meta Values

- *csrf_secret* A byte string which is the master key by which we encode all values. Set to a sufficiently long string of characters that is difficult to guess or bruteforce (recommended at least 16 characters) for example the output of `os.urandom(16)`.
- *csrf_time_limit* if *None*, tokens last forever (not recommended.) Otherwise, set to a *datetime.timedelta* that will define how long CSRF tokens are valid for. Defaults to 30 minutes.
- *csrf_context* This should be a *request.session*-style object. Usually given in the Form constructor.

Example

```
from wtforms.csrf.session import SessionCSRF
from datetime import timedelta

class MyBaseForm(Form):
    class Meta:
        csrf = True
        csrf_class = SessionCSRF
        csrf_secret = b'EPj00jpfj8Gx1SjnyLxwBBSQfnQ9DJYe0Ym'
        csrf_time_limit = timedelta(minutes=20)

class Registration(MyBaseForm):
    name = StringField()

def view(request):
    form = Registration(request.POST, meta={'csrf_context': request.session})
    # rest of view here
```

Note that *request.session* is passed as the *csrf_context* override to the meta info, this is so that the CSRF token can be stored in your session for comparison on a later request.

Example Integration

WTForms primitives are designed to work with a large variety of frameworks, and as such sometimes things seem like they are more work to use, but with some smart integration, you can actually clean up your code substantially.

For example, if you were going to integrate with *Flask*, and wanted to use the *SessionCSRF* implementation, here's one way to get the CSRF context to be available without passing it all the time:

```
from flask import session
from wtforms.csrf.session import SessionCSRF
```

(continues on next page)

(continued from previous page)

```
class MyBaseForm(Form):
    class Meta:
        csrf = True
        csrf_class = SessionCSRF
        csrf_secret = app.config['CSRF_SECRET_KEY']

        @property
        def csrf_context(self):
            return session
```

Now with any subclasses of `MyBaseForm`, you don't need to pass in the csrf context, and on top of that, we grab the secret key out of your normal app configuration.

2.7 Internationalization (i18n)

Localizing strings in WTForms is a topic that frequently comes up. In WTForms, the majority of messages that are transmitted are provided by you, the user. However, there is support for translating some of the *built-in* messages in WTForms (such as errors which occur during data coercion) so that the user can make sure the user experience is consistent.

2.7.1 Translating user-provided messages

This is not actually any specific feature in WTForms, but because the question is asked so frequently, we need to address it here: **WTForms does -not- translate any user-provided strings.**

This is not to say they can't be translated, but that it's up to you to deal with providing a translation for any passed-in messages. WTForms waits until the last moment (usually validation time) before doing anything with the passed in message (such as interpolating strings) thus giving you the opportunity to e.g. change your locale before validation occurs, if you are using a suitable "lazy proxy".

Here's a simple example of how one would provide translated strings to WTForms:

```
from somelibrary import ugettext_lazy as _
from wtforms import Form, StringField, IntegerField, validators as v

class RegistrationForm(Form):
    name = StringField(_('Name'), [v.InputRequired(_('Please provide your name'))])
    age = IntegerField(
        _('Age'),
        [v.NumberRange(min=12, message=_('Must be at least %(min)d years old.'))]
    )
```

The field label is left un-perturbed until rendering time in a template, so you can easily provide translations for field labels if so desired. In addition, validator messages with format strings are not interpolated until the validation is run, so you can provide localization there as well.

2.7.2 Translating built-in messages

There are some messages in WTForms which are provided by the framework, namely default validator messages and errors occurring during the processing (data coercion) stage. For example, in the case of the `IntegerField` above, if someone entered a value which was not valid as an integer, then a message like “Not a valid integer value” would be displayed.

Using the built-in translations provider

WTForms now includes a basic translations provider which uses the stdlib `gettext` module to localize strings based on locale information distributed with the package. Localizations for several languages are included, and we hope that soon there will be more submitted.

To use the builtin translations provider, simply pass locale languages as locales in the *meta* section of the [constructor](#) of your form:

```
form = MyForm(request.form, meta={'locales': ['en_US', 'en']})
```

Alternately, if you are localizing application-wide you can define locales at meta-level in a subclass of `Form`:

```
class MyBaseForm(Form):
    class Meta:
        locales = ['es_ES', 'es']
```

Now simply have all your forms be a subclass of `MyBaseForm` and you will have all your default messages output in spanish.

Writing your own translations provider

For this case, we provide the ability to give a translations object on a subclass of `Form`, which will then be called to translate built-in strings.

An example of writing a simple translations object:

```
from mylibrary import ugettext, ungettext
from wtforms import Form

class MyTranslations(object):
    def gettext(self, string):
        return ugettext(string)

    def ngettext(self, singular, plural, n):
        return ungettext(singular, plural, n)

class MyBaseForm(Form):
    def _get_translations(self):
        return MyTranslations()
```

You would then use this new base `Form` class as the base class for any forms you create, and any built-in messages from WTForms will be passed to your `gettext/ngettext` implementations.

You control the object’s constructor, its lifecycle, and everything else about it, so you could, for example, pass the locale per-form instantiation to the translation object’s constructor, and anything else you need to do for translations to work for you.

3.1 Changes

3.1.1 Version 3.0.2

Unreleased

- Delayed import of `email_validator`. #727

3.1.2 Version 3.0.1

Released 2021-12-23

- Fixed `DateTimeField` and other similar fields can handle multiple formats. #720 #721
- Stop support for python 3.6 #722

3.1.3 Version 3.0.0

Released 2021-11-07

- Fixed `RadioField` validators. #477 #615
- `populate_obj()` always calls `setattr()` #675
- WTForms has a new logo. #569 #689
- Fixed `RadioField` `render_kw` rendering. #490 #628 #688
- Support for optgroups in `SelectField` and `SelectMultipleField`. #656 #667
- Minor documentation fix. #701
- Custom separators for `FieldList`. #681 #694
- `DateTimeField`, `DateField` and `TimeField` support time formats that removes leading zeros. #703
- Refactoring: split `fields/core.py` and `fields/simple.py` #710

3.1.4 Version 3.0.0a1

Released 2020-11-23

- Drop support for Python < 3.6. [#554](#)
- `StringField` sets data to None when form data is empty and an initial value was not provided. Although it previously set an empty string, None is consistent with the behavior of other fields. [#355](#)
- Specified version of Babel required for setup to avoid errors. [#430](#)
- Replaced use of `getattr/setattr` with regular variable access. [#482](#)
- `ValueError` raised by a validator are handled like regular exceptions. Validators need to raise `ValidationError` or `StopValidation` to make a validation fail. [#445](#)
- `SelectField`, `SelectMultipleField` and `RadioField` `choices` parameter can be a callable. [#608](#)
- Choices shortcut for `SelectMultipleField`. [#603](#) [#605](#)
- Forms can have form-level errors. [#55](#) [#595](#)
- Implemented `MonthField`. [#530](#) [#593](#)
- Filters can be inline. `form.BaseForm.process()` takes a `extra_filters` parameter. [#128](#) [#592](#)
- Fields can be passed the `name` argument to use a HTML name different than their Python name. [#205](#), [#601](#)
- Render attribute names like `for_` and `class_` are normalized consistently so later values override those specified earlier. [#449](#), [#596](#)
- Flags can take non-boolean values. [#406](#) [#467](#)
- Widgets are HTML5 by default. [#594](#) [#614](#)
- Fixed a bug when the `SelectField` `choices` are list of strings. [#598](#)
- Error messages standardization. [#613](#) [#620](#) [#626](#) [#627](#)
- `SelectMultipleField` `validate_choice` bugfix. [#606](#) [#642](#)
- Fixed `SelectMultipleField` validation when using `choices` list shortcut. [#612](#) [#661](#)

3.1.5 Version 2.3.3

Released 2020-07-30

- This release includes the translation files that were missing in the 2.3.2 release. [#641](#)

3.1.6 Version 2.3.2

Released 2020-07-29

- Fixed a bug with `SelectField` `choices` shortcut at form submission. [#598](#)[#639](#)

3.1.7 Version 2.3.1

Released 2020-04-22

- All modules in `wtforms.ext` show a deprecation warning on import. They will be removed in version 3.0.
- Fixed a bug when `SelectField` choices is `None`. #572#585
- Restored `HTMLString` and `escape_html` as aliases for MarkupSafe functions. Their use shows a `DeprecationWarning`. #581, #583
- `Form.validate` takes an `extra_validators` parameter, mapping field names to lists of extra validator functions. This matches `BaseForm.validate`. #584
- Update locale catalogs.

3.1.8 Version 2.3.0

Released 2020-04-21

- Drop support for Python 2.6, 3.3, and 3.4.
- `SelectField` uses `list()` to construct a new list of choices. #475
- Permitted underscores in `HostnameValidation`. #463
- `URL` validator now allows query parameters in the URL. #523, #524
- Updated `false_values` param in `BooleanField` docs. #483, #485
- Fixed broken format string in Arabic translation #471
- Updated French and Japanese translations. #506#514
- Updated Ukrainian translation. #433
- `FieldList` error list keeps entries in order for easier identification of which fields had errors. #257, #407
- `Length` gives a more helpful error message when `min` and `max` are the same value. #266
- `SelectField` no longer coerces `None` to `"None"` allowing use of `"None"` as an option. #289, #288
- The `TextArea` widget prepends a `\r\n` newline when rendering to account for browsers stripping an initial line for display. This does not affect the value. #238, #395
- `HTML5 IntegerField` and `RangeInput` don't render the `step="1"` attribute by default. #343
- `aria_args` are rendered the same way as `data_args`, by converting underscores to hyphens. `aria_describedby="name-help"` becomes `aria-describedby="name-help"`. #239, #389
- Added a `check_validators` method to `Field` which checks if the given validators are both callable, and not classes. #298#410
- `form.errors` is not cached and will update if an error is appended to a field after access. #568
- `NumberRange` correctly handle NaN values. #505, #548
- `IntegerField` checks input type when processing data. #451
- Added a parameter to `SelectField` to skip choice validation. #434, #493
- Choices which name and data are the same do not need to use tuples. #526
- Added more documentation on HTML5 fields. #326#409

- HTML is escaped using MarkupSafe instead of the previous internal implementation. `escape_html()` is removed, replaced by `markupsafe.escape()`. `HTMLString` is removed, replaced by `markupsafe.Markup`. #400
- Fixed broken IPv6 validator, validation now uses the `ipaddress` package. #385, #403
- Label text is escaped before rendering. #315, #375
- Email validation is now handled by an optional library, `email_validator`. #429

3.1.9 Version 2.2.1

Released 2018-06-07

- `StringField` only sets `data = ''` when form data is empty and an initial value was not provided. This fixes an issue where the default value wasn't rendered with the initial form. #291#401, #355

3.1.10 Version 2.2

Released 2018-06-02

- Merged new and updated translations from the community.
- Passing `data_args` to render a field converts all the underscores to hyphens when rendering the HTML attribute, not just the first one. `data_foo_bar` becomes `data-foo-bar`. #248
- The `UUID` validator uses the `uuid.UUID` class instead of a regex. #251
- `SelectField` copies the list of `choices` passed to it so modifying an instance's choices will not modify the global form definition. #286
- Fields call `process_formdata()` even if the raw data is empty. #280
- Added a `MultipleFileField` to handle a multi-file input. `FileField` continues to handle only one value. The underlying `FileInput` widget gained a `multiple` argument. #281
- `SelectField` choices can contain HTML (MarkupSafe Markup object or equivalent API) and will be rendered properly. #302
- `fields.TimeField` and `html5.TimeField` were added. #254
- Improved `Email`. Note that it is still unreasonable to validate all emails with a regex and you should prefer validating by actually sending an email. #294
- Widgets render the `required` attribute when using a validator that provides the 'required' flag, such as `DataRequired`. #361
- Fix a compatibility issue with SQLAlchemy 2.1 that caused `QuerySelectField` to fail with `ValueError: too many values to unpack`. #391

3.1.11 Version 2.1

Released 2015-12-15

- Added `render_kw` to allow default rendering time options.
- Updated / added a number of localizations.
- Updated docs.
- Allow widgets to set flags.

3.1.12 Version 2.0.2

Released 2015-01-18

- Added more localizations and updated some.
- Validators for email and URL can validate IDNA-encoded domain names and new TLDs.
- Better DeprecationWarnings.
- Support localization files in `/usr/share/locale` for distro packaging.

3.1.13 Version 2.0.1

Released 2014-07-01

- Update wheel install to conditionally install `ordereddict` for Python 2.6.
- Doc improvements.

3.1.14 Version 2.0

Released 2014-05-20

- Add new class `Meta` paradigm for much more powerful customization of WTForms.
- Move `i18n` into core. Deprecate `wtforms.ext.i18n`.
- Move CSRF into core. Deprecate `wtforms.ext.csrf`.
- Fix issue rendering `SelectFields` with `value=True`.
- Make `DecimalField` able to use babel locale-based number formatting.
- Drop Python 3.2 support (Python3 support for 3.3+ only).
- Passing `attr=False` to WTForms widgets causes the value to be ignored.
- Unique validator in `wtforms.ext.sqlalchemy` has been removed.

3.1.15 Version 1.0.5

Released 2013-09-10

- Fix a bug in validators which causes translations to happen once then clobber any future translations.
- `ext.sqlalchemy` and `ext.appengine` minor cleanups / deprecation.
- Allow blank string and the string `false` to be considered false values for `BooleanField` (configurable). This is technically a breaking change, but it is not likely to affect the majority of users adversely.
- `ext.i18n` form allows passing `LANGUAGES` to the constructor.

3.1.16 Version 1.0.4

Released 2013-04-28

- Add widgets and field implementations for HTML5 specialty input types.
- `ext.appengine` add NDB support.
- Add translations for Korean, Traditional Chinese.

3.1.17 Version 1.0.3

Released 2013-01-24

- Tests complete in python 3.2/3.3.
- Localization for ru, fr.
- Minor fixes in documentation for clarity.
- `FieldList` now can take validators on the entire `FieldList`.
- Fix issue with `ext.sqlalchemy QuerySelectField`.
- Fix issue in `ext.sqlalchemy ColumnDefault` conversion.
- `ext.sqlalchemy` supports Enum type.
- Field class now allows traversal in Django 1.4 templates.

3.1.18 Version 1.0.2

Released 2012-08-24

- We now support Python 2.x and 3.x on the same codebase, thanks to a lot of hard work by Vinay Sajip.
- Add in ability to convert relationships to `ext.sqlalchemy model_form`.
- Built-in localizations for more languages.
- Distinguish Required validator into `InputRequired` and `DataRequired`.
- Better IP address validation, including IPv6 support.
- `AnyOf` / `NoneOf` now work properly formatting other datatypes than strings.
- `Optional` validator can optionally pass through whitespace.

3.1.19 Version 1.0.1

Released 2012-02-29

- Fixed issues related to building for Python 3 and Python pre-releases.
- Add `object_data` to fields to get at the originally passed data.

3.1.20 Version 1.0

Released 2012-02-28

- Output HTML5 compact syntax by default.
- Substantial code reorg, cleanup, and test improvements.
- Added `ext.csrf` for a way to implement CSRF protection.
- `ext.sqlalchemy` supports PGInet, MACADDR, and UUID field conversion.
- `ext.sqlalchemy` supports callable defaults.
- `ext.appengine model_form` now supports generating forms with the same ordering as model.
- `ext.appengine ReferencePropertyField` now gets `get_label` like the other ORM fields.
- Add localization support for WTForms built-in messages.
- Python 3 support (via 2to3).
- An empty label string can be specified on fields if desired.
- Option widget can now take kwargs customization.
- Field subclasses can provide default validators as a class property.
- `DateTimeField` can take time in microseconds.
- Numeric fields all set `.data` to `None` on coercion error for consistency.

3.1.21 Version 0.6.3

Released 2011-04-24

- Documentation: Substantial documentation improvements, including adding Crash Course as a Sphinx document.
- `ext.django QuerySetSelectField` and `ModelSelectField` now accept `get_label` similar to sqlalchemy equivalents.
- `ext.appengine model_form` fixes for `FloatField`, `TimeField`, and `DateTimeField`.
- `ext.appengine ReferencePropertyField` now properly stores model object, not key.

3.1.22 Version 0.6.2

Released 2011-01-22

- `ext.appengine` various field fixes.
- `ext.appengine model_form` changes.
- Fix issue in `Optional` with non-string input.
- Make numeric fields more consistent.
- Improve test coverage substantially.

3.1.23 Version 0.6.1

Released 2010-09-17

- `ext.appengine ReferencePropertyField`.
- Dateutil fields render issue, and consistency issue.
- `Optional` validator failed when `raw_data` was absent
- Documentation: docs now mention HTML escaping functionality.
- Add preliminary support for providing a translations object that can translate built-in validation and coercion errors.

3.1.24 Version 0.6

Released 2010-04-25

- HTML is now marked as safe (using `__html__`) so that compatible templating engines will not auto-escape it.
- `Field._default` is now `Field.default`.
- All fields now have a `raw_data` property.
- Fields which are select fields (including those in `.ext`) can be iterated to produce options, and have an `option_widget` kwarg.
- Minor bugfixes and cleanup in `FieldList`, `Select(Multiple)Field`, `QuerySelectField` to address behavioral consistency.
- Added `FloatField`, based on `IntegerField`.
- `ext.appengine` now supports `FloatProperty` and `GeoPtProperty`.
- `ext.sqlalchemy QueryMultipleSelectField` changed to `QuerySelectMultipleField`.

3.1.25 Version 0.5

Released 2010-02-13

- Added a `BaseForm` class which provides the core processing and validation functionality of `Form` without requiring declarative subclassing.
- Field labels now default to a humanized field name.
- Fields now have a `short_name` property which is the un-prefixed name.
- `DecimalField` now rounds values for display without float coercion. See docs for details on how to format decimals.
- `ext.sqlalchemy.fields` now has an additional `QuerySelectMultipleField`, and all fields can now support multiple-column primary keys.
- `ext.sqlalchemy.orm` contains tools for making forms from ORM models.
- Added `ext.dateutil` for flexible date-time parsing.
- Added `ext.appengine` contributed by Rodrigo Moraes.
- Added `AnyOf` and `NoneOf` validators.

3.1.26 Version 0.4

Released 2009-10-10

- Fields have much greater control over input processing. Filters have been added to implement a simple way to transform input data.
- Added fields that encapsulate advanced data structures such as dynamic lists or child forms for more powerful field composing.
- Fields now use widgets for rendering.
- All built-in validators have been converted to classes to clean up the code.
- `Form.auto_populate` and `Field.populate` were renamed to `populate_obj` to clarify that they populate another object, not the `Form` or `Field`. This is an API breaking change.
- Dropped support for Python 2.3.

3.1.27 Version 0.3.1

Released 2009-01-24

- Several fixes were made to the code and tests to make WTForms compatible with Python 2.3/2.4.
- `Form`'s properties can now be accessed via dictionary-style access such as `form['author']`. This also has the intended effect of making variable lookups in Django templates more reliable.
- `Form` and `Field` construction changes: `Form` now uses a metaclass to handle creating its `_unbound_fields` property, and `Field` construction now gives an instance of the new `UnboundField` class instead of using a partial function application. These are both internal changes and do not change the API.

3.1.28 Version 0.3

Released 2009-01-18

- Fields are now responsible for their own validation, instead of mostly relying on `Form`. There are also new `pre_validate` and `post_validate` hooks on subfields, adding a great deal of flexibility when dealing with field-level validation. Note that this is an API breaking change if you have any subfields that override `Field.validate`. These will need to be updated to use the new hooks.
- `process_data` no longer accepts the `has_formdata` parameter.
- At form instantiation time, `process_data` will be called only once for each field. If a model object is provided which contains the property, then this value is used. Otherwise, a keyword argument if specified is used. Failing that, the field's default value is used.
- If any form data is sent, `process_formdata` will be called after `process_data` for each field. If no form data is available for the given field, it is called with an empty list.
- `wtforms.ext.django` has been overhauled, both to mirror features and changes of the Django 1.0 release, and to add some useful fields for working with Django ORM data in forms.
- The `checker` keyword argument to `SelectField`, `SelectMultipleField`, and `RadioField` has been re-named to `coerce` to reflect the actual functionality of this callable.

3.1.29 Version 0.2

Released 2009-01-13

- We have documentation and unit tests!
- Fields now have a `flags` property which contain boolean flags that are set either by the field itself or validators being specified on a field. The flags can then be used in checks in template or Python code.
- Changed the way fields take parameters, they are no longer quasi magic. This is a breaking change. Please see the documentation for the new syntax.
- Added optional description argument to `Field`, accessible on the field as `description`. This provides an easy way to define e.g. help text in the same place as the form.
- Added new semantics for validators which can stop the validation chain, with or without errors.
- Added a regexp validator, and removed the `not_empty` validator in favour of two validators, `optional` and `required`. The new validators allow control over the validation chain in addition to checking emptiness.
- Renamed `wtforms.contrib` to `wtforms.ext` and reorganised `wtforms.ext.django`. This is a breaking change if you were using the Django extensions, but should only require changing your imports around a little.
- Better support for other frameworks such as Pylons.

3.1.30 Version 0.1

Released 2008-07-25

- Initial release.

3.2 What's New in WTForms 3

3.2.1 New Features

WTForms 3 is something something TODO

3.2.2 Past Major Releases

WTForms 2

WTForms 2 was the first major version bump since WTForms 1.0. Coming with it are a number of major changes that allow far more customization of core WTForms features. This is done to make WTForms even more capable when working along with companion libraries.

New Features

- *Class Meta* paradigm allows customization of many aspects of WTForms.
- *CSRF* and *i18n* are core features not needing extensions anymore.
- Widget rendering changes:
 - Passing `<attribute name>=False` to WTForms widget rendering is now ignored, making it easier to deal with boolean HTML attributes.
 - Creating an html attribute `data-foo` can be done by passing the keyword `data_foo` to the widget.

Deprecated API's

These API's still work, but in most cases will cause a `DeprecationWarning`. The deprecated API's will be removed in WTForms 3.0, so write code against the new API's unless it needs to work across both WTForms 1.x and 2.x

- **Core**
 - `Form._get_translations` Use `Meta.get_translations` instead.
 - The `TextField` alias for `StringField` is deprecated.
 - `wtforms.validators.Required` is now `wtforms.validators.DataRequired`
 - `wtforms.fields._unset_value` is now `wtforms.utils.unset_value`
- **WTForms Extensions** All the extensions are being deprecated. We feel like the extensions we had would actually benefit from being pulled outside the WTForms package, because it would allow them to have a separate release schedule that suits their companion libraries.
 - `wtforms.ext.appengine` Is deprecated, see [WTForms-Appengine](#)
 - `wtforms.ext.csrf` CSRF protection is now *built in*

- `wtforms.ext.dateutil` Is deprecated, but does not have a new home yet.
- `wtforms.ext.django` Is deprecated. See [WTForms-Django](#)
- `wtforms.ext.i18n` is now *built in*
- `wtforms.ext.sqlalchemy` Is deprecated, look at [WTForms-Alchemy \(docs\)](#)

Low-level Changes

Most of these changes shouldn't affect the typical library user, however we are including these changes for completeness for those who are creating companion libraries to WTForms.

- `BaseForm._fields` is now an `OrderedDict`, not a plain dict.
- `FormMeta` now manages an attribute called `_wtforms_meta` which is a subclass of any class `Meta` defined on ancestor form classes.
- A new keyword-param called simply `data=` to the Form constructor has been added and positioned as the place where soon we will be able to accept structured data which is neither formdata, object data, or defaults. Currently this parameter is merged with the `kwargs`, but the intention is to handle other structured data (think JSON).
- *Filters* on fields stop on the first `ValueError`, instead of continuing on to the next one.

3.3 BSD-3-Clause License

Copyright 2008 WTForms

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

3.4 Contributing to WTForms

WTForms is an open-source library and changing and evolving over time. To that end, we are supported by the contributions of many people in the python community.

3.4.1 How to Contribute

WTForms is now on [GitHub](#), so all contributions should be either associated with a pull request or with a ticket & patch.

3.4.2 Contribution Guidelines

Code submitted should:

- Be formatted according to the [PEP8](#) style guideline **except** that it does not need to confirm to the 79-column limit requirement of the guideline.
- Have tests
 - Unless it's a bugfix, it should pass existing tests.
 - New classes or methods should mean new unit tests or extending existing tests.
 - Bugfixes can probably do with a regression test too (a test that would fail without this fix)
- Use naming schemes consistent with WTForms conventions
- Work on all versions of Python that WTForms currently supports (and python3 without needing 2to3). Take advantage of [Travis-CI](#) for running tests on all supported Python versions.

3.4.3 Note on API compatibility

WTForms is a very small library, but yet it's possible to break API compatibility pretty easily. We are okay with breaking API compatibility for compelling features or major changes that we feel are worthwhile inclusions to the WTForms core, but realize that any API compatibility break will delay the inclusion of your ticket to the next major release.

Some examples of API compatibility breaks include:

- Adding new attributes or methods to the base Form class
- Modifying the number of required arguments of core methods like `process()`
- Changing the default behavior of a field.

However, it is still possible to add new features to WTForms without breaking API compatibility. For example, if one were looking to add Babel locale support to `DecimalField`, it could be done so long as by default, `DecimalField` behaved the same as it did before. This could look something like:

1. Add a keyword arg `use_locale` to the constructor
2. Make the keyword default to `False` so the behavior without this arg is identical to the previous behavior.
3. Add your functionality and make sure all existing `DecimalField` tests work unchanged (and of course add new tests for the new functionality).

PYTHON MODULE INDEX

W

- `wtforms.csrf`, [43](#)
- `wtforms.csrf.core`, [44](#)
- `wtforms.csrf.session`, [48](#)
- `wtforms.fields`, [21](#)
- `wtforms.form`, [15](#)
- `wtforms.meta`, [41](#)
- `wtforms.validators`, [32](#)

Symbols

[__call__\(\) \(wtforms.fields.Field method\), 23](#)
[__contains__\(\) \(wtforms.form.BaseForm method\), 20](#)
[__contains__\(\) \(wtforms.form.Form method\), 17](#)
[__delitem__\(\) \(wtforms.form.BaseForm method\), 20](#)
[__getitem__\(\) \(wtforms.fields.FieldList method\), 30](#)
[__getitem__\(\) \(wtforms.form.BaseForm method\), 20](#)
[__html__\(\) \(wtforms.fields.Field method\), 23](#)
[__init__\(\) \(wtforms.csrf.core.CSRFTokenField method\), 44](#)
[__init__\(\) \(wtforms.fields.Field method\), 21](#)
[__init__\(\) \(wtforms.form.BaseForm method\), 19](#)
[__init__\(\) \(wtforms.form.Form method\), 15](#)
[__iter__\(\) \(wtforms.fields.FieldList method\), 30](#)
[__iter__\(\) \(wtforms.form.BaseForm method\), 20](#)
[__iter__\(\) \(wtforms.form.Form method\), 16](#)
[__len__\(\) \(wtforms.fields.FieldList method\), 30](#)
[__setitem__\(\) \(wtforms.form.BaseForm method\), 20](#)
[_value\(\) \(wtforms.csrf.core.CSRFTokenField method\), 45](#)

A

[AnyOf \(class in wtforms.validators\), 35](#)
[append_entry\(\) \(wtforms.fields.FieldList method\), 29](#)

B

[BaseForm \(class in wtforms.form\), 19](#)
[bind_field\(\) \(wtforms.meta.DefaultMeta method\), 42](#)
[BooleanField \(class in wtforms.fields\), 25](#)
[build_csrf\(\) \(wtforms.meta.DefaultMeta method\), 42](#)

C

[cache_translations \(wtforms.meta.DefaultMeta attribute\), 42](#)
[CheckboxInput \(class in wtforms.widgets\), 38](#)
[ColorInput \(class in wtforms.widgets\), 38](#)
[CSRF \(class in wtforms.csrf.core\), 45](#)
[csrf \(wtforms.meta.DefaultMeta attribute\), 41](#)
[csrf_class \(wtforms.meta.DefaultMeta attribute\), 42](#)
[csrf_field_name \(wtforms.meta.DefaultMeta attribute\), 42](#)
[CSRFTokenField \(class in wtforms.csrf.core\), 44](#)

[current_token \(wtforms.csrf.core.CSRFTokenField attribute\), 45](#)

D

[data \(wtforms.fields.Field attribute\), 23](#)
[data \(wtforms.form.BaseForm attribute\), 19](#)
[data \(wtforms.form.Form attribute\), 16](#)
[DataRequired \(class in wtforms.validators\), 32](#)
[DateField \(class in wtforms.fields\), 25](#)
[DateInput \(class in wtforms.widgets\), 38](#)
[DateTimeField \(class in wtforms.fields\), 25](#)
[DateTimeInput \(class in wtforms.widgets\), 38](#)
[DateTimeLocalField \(class in wtforms.fields\), 25](#)
[DateTimeLocalInput \(class in wtforms.widgets\), 38](#)
[DecimalField \(class in wtforms.fields\), 25](#)
[DecimalRangeField \(class in wtforms.fields\), 26](#)
[default \(wtforms.fields.Field attribute\), 24](#)
[DefaultMeta \(class in wtforms.meta\), 41](#)
[description \(wtforms.fields.Field attribute\), 24](#)

E

[Email \(class in wtforms.validators\), 33](#)
[EmailField \(class in wtforms.fields\), 26](#)
[EmailInput \(class in wtforms.widgets\), 38](#)
[entries \(wtforms.fields.FieldList attribute\), 30](#)
[EqualTo \(class in wtforms.validators\), 33](#)
[errors \(wtforms.fields.Field attribute\), 22](#)
[errors \(wtforms.form.BaseForm attribute\), 20](#)
[errors \(wtforms.form.Form attribute\), 16](#)

F

[Field \(class in wtforms.fields\), 21](#)
[field_class \(wtforms.csrf.core.CSRF attribute\), 46](#)
[field_id \(wtforms.fields.Label attribute\), 32](#)
[FieldList \(class in wtforms.fields\), 29](#)
[FileField \(class in wtforms.fields\), 26](#)
[FileInput \(class in wtforms.widgets\), 38](#)
[filters \(wtforms.fields.Field attribute\), 25](#)
[Flags \(class in wtforms.fields\), 32](#)
[flags \(wtforms.fields.Field attribute\), 25](#)
[FloatField \(class in wtforms.fields\), 26](#)
[Form \(class in wtforms.form\), 15](#)

`form_errors` (*wtforms.form.Form* attribute), 16
`FormField` (class in *wtforms.fields*), 29

G

`generate_csrf_token()` (*wtforms.csrf.core.CSRF* method), 45
`get_translations()` (*wtforms.meta.DefaultMeta* method), 42
`gettext()` (*wtforms.fields.Field* method), 24

H

`HiddenField` (class in *wtforms.fields*), 28
`HiddenInput` (class in *wtforms.widgets*), 38
`html_params()` (in module *wtforms.widgets*), 40

I

`id` (*wtforms.fields.Field* attribute), 24
`Input` (class in *wtforms.widgets*), 38
`InputRequired` (class in *wtforms.validators*), 33
`IntegerField` (class in *wtforms.fields*), 26
`IntegerRangeField` (class in *wtforms.fields*), 26
`IPAddress` (class in *wtforms.validators*), 34

L

`Label` (class in *wtforms.fields*), 32
`label` (*wtforms.fields.Field* attribute), 24
`Length` (class in *wtforms.validators*), 34
`ListWidget` (class in *wtforms.widgets*), 38
`locales` (*wtforms.meta.DefaultMeta* attribute), 42

M

`MacAddress` (class in *wtforms.validators*), 34
`meta` (*wtforms.fields.Field* attribute), 25
`meta` (*wtforms.form.Form* attribute), 16
module
 wtforms.csrf, 43
 wtforms.csrf.core, 44
 wtforms.csrf.session, 48
 wtforms.fields, 21
 wtforms.form, 15
 wtforms.meta, 41
 wtforms.validators, 32
`MonthField` (class in *wtforms.fields*), 26
`MonthInput` (class in *wtforms.widgets*), 38
`MultipleFileField` (class in *wtforms.fields*), 26

N

`name` (*wtforms.fields.Field* attribute), 24
`ngettext()` (*wtforms.fields.Field* method), 24
`NoneOf` (class in *wtforms.validators*), 35
`NumberInput` (class in *wtforms.widgets*), 39
`NumberRange` (class in *wtforms.validators*), 34

O

`object_data` (*wtforms.fields.Field* attribute), 23
`Optional` (class in *wtforms.validators*), 34

P

`PasswordField` (class in *wtforms.fields*), 28
`PasswordInput` (class in *wtforms.widgets*), 39
`pop_entry()` (*wtforms.fields.FieldList* method), 30
`populate_obj()` (*wtforms.csrf.core.CSRFTokenField* method), 45
`populate_obj()` (*wtforms.form.Form* method), 16
`post_validate()` (*wtforms.fields.Field* method), 22
`pre_validate()` (*wtforms.csrf.core.CSRFTokenField* method), 45
`pre_validate()` (*wtforms.fields.Field* method), 22
`process()` (*wtforms.csrf.core.CSRFTokenField* method), 45
`process()` (*wtforms.fields.Field* method), 22
`process()` (*wtforms.form.BaseForm* method), 20
`process_data()` (*wtforms.fields.Field* method), 23
`process_errors` (*wtforms.fields.Field* attribute), 24
`process_formdata()` (*wtforms.fields.Field* method), 23

R

`RadioField` (class in *wtforms.fields*), 26
`RadioInput` (class in *wtforms.widgets*), 39
`RangeInput` (class in *wtforms.widgets*), 39
`raw_data` (*wtforms.fields.Field* attribute), 23
`Regexp` (class in *wtforms.validators*), 35
`render_field()` (*wtforms.meta.DefaultMeta* method), 43

S

`SearchField` (class in *wtforms.fields*), 28
`SearchInput` (class in *wtforms.widgets*), 39
`Select` (class in *wtforms.widgets*), 39
`SelectField` (class in *wtforms.fields*), 27
`SelectMultipleField` (class in *wtforms.fields*), 28
`SessionCSRF` (class in *wtforms.csrf.session*), 48
`setup_form()` (*wtforms.csrf.core.CSRF* method), 45
`short_name` (*wtforms.fields.Field* attribute), 24
`StopValidation` (class in *wtforms.validators*), 32
`StringField` (class in *wtforms.fields*), 28
`SubmitField` (class in *wtforms.fields*), 28
`SubmitInput` (class in *wtforms.widgets*), 39

T

`TableWidget` (class in *wtforms.widgets*), 39
`TelField` (class in *wtforms.fields*), 28
`TelInput` (class in *wtforms.widgets*), 39
`text` (*wtforms.fields.Label* attribute), 32
`TextArea` (class in *wtforms.widgets*), 39
`TextAreaField` (class in *wtforms.fields*), 28

`TextInput` (*class in wtforms.widgets*), 39
`TimeField` (*class in wtforms.fields*), 28
`TimeInput` (*class in wtforms.widgets*), 39
`type` (*wtforms.fields.Field attribute*), 24

U

`URL` (*class in wtforms.validators*), 35
`URLField` (*class in wtforms.fields*), 28
`URLInput` (*class in wtforms.widgets*), 39
`UUID` (*class in wtforms.validators*), 35

V

`validate()` (*wtforms.fields.Field method*), 22
`validate()` (*wtforms.form.BaseForm method*), 20
`validate()` (*wtforms.form.Form method*), 16
`validate_csrf_token()` (*wtforms.csrf.core.CSRF method*), 45
`ValidationError` (*class in wtforms.validators*), 32

W

`WeekInput` (*class in wtforms.widgets*), 39
`widget` (*wtforms.fields.Field attribute*), 24
`wrap_formdata()` (*wtforms.meta.DefaultMeta method*), 42
`wtforms.csrf`
 module, 43
`wtforms.csrf.core`
 module, 44
`wtforms.csrf.session`
 module, 48
`wtforms.fields`
 module, 21
`wtforms.form`
 module, 15
`wtforms.meta`
 module, 41
`wtforms.validators`
 module, 32