Widgets

A widget is Django's representation of an HTML input element. The widget handles the rendering of the HTML, and the extraction of data from a GET/POST dictionary that corresponds to the widget.

The HTML generated by the built-in widgets uses HTML5 syntax, targeting `<!DOCTYPE html>`. For example, it uses boolean attributes such as `checked` rather than the XHTML style of `checked='checked'`.

Tip

Widgets should not be confused with the form fields. Form fields deal with the logic of input validation and are used directly in templates. Widgets deal with rendering of HTML form input elements on the web page and extraction of raw submitted data. However, widgets do need to be assigned to form fields.

Specifying widgets

Whenever you specify a field on a form, Django will use a default widget that is appropriate to the type of data that is to be displayed. To find which widget is used on which field, see the documentation about Built-in Field classes.

However, if you want to use a different widget for a field, you can use the `widget` argument on the field definition. For example:

```
from django import forms

class CommentForm(forms.Form):
    name = forms.CharField()
    url = forms.URLField()
    comment = forms.CharField(widget=forms.Textarea)
```

This would specify a form with a comment that uses a larger `Textarea` widget, rather than the default `TextInput` widget.

Setting arguments for widgets

Many widgets have optional extra arguments; they can be set when defining the widget on the field. In the following example, the `years` attribute is set for a `SelectDateWidget`:

```
from django import forms

BIRTH_YEAR_CHOICES = ['1980', '1981', '1982']
FAVORITE_COLORS_CHOICES = [
    ('blue', 'Blue'),
    ('green', 'Green'),
    ('black', 'Black'),
]

class SimpleForm(forms.Form):
    birth_year = forms.DateField(widget=forms.SelectDateWidget(years=BIRTH_YEAR_CHOICES))
    favorite_colors = forms.MultipleChoiceField(
        required=False,
        widget=forms.CheckboxSelectMultiple,
        choices=FAVORITE_COLORS_CHOICES,
    )
```

See the Built-in widgets for more information about which widgets are available and which arguments they accept.

Widgets inheriting from the `Select` widget

Widgets inheriting from the `Select` widget deal with choices. They present the user with a list of options to choose from. The different widgets present this choice differently; the `Select` widget itself uses a `<select>` HTML list representation, while `RadioSelect` uses radio buttons.

`Select` widgets are used by default on `ChoiceField` fields. The choices displayed on the widget are inherited from the `ChoiceField` and changing `ChoiceField.choices` will update `Select.choices`. For example:

```
>>> from django import forms
>>> CHOICES = [('1', 'First'), ('2', 'Second')]
>>> choice_field = forms.ChoiceField(widget=forms.RadioSelect, choices=CHOICES)
>>> choice_field.choices
[('1', 'First'), ('2', 'Second')]
>>> choice_field.widget.choices
[('1', 'First'), ('2', 'Second')]
>>> choice_field.widget.choices = []
>>> choice_field.choices = [('1', 'First and only')]
>>> choice_field.widget.choices
[('1', 'First and only')]
```

Widgets which offer a `choices` attribute can however be used with fields which are not based on choice – such as a `CharField` – but it is recommended to use a `ChoiceField`-based field when the choices are inherent to the model and not just the representational widget.

Customizing widget instances

When Django renders a widget as HTML, it only renders very minimal markup - Django doesn't add class names, or any other widget-specific attributes. This means, for example, that all `TextInput` widgets will appear the same on your Web pages.

There are two ways to customize widgets: per widget instance and per widget class.

Styling widget instances

If you want to make one widget instance look different from another, you will need to specify additional attributes at the time when the widget object is instantiated and assigned to a form field (and perhaps add some rules to your CSS files).

For example, take the following form:

```
from django import forms

class CommentForm(forms.Form):
    name = forms.CharField()
    url = forms.URLField()
    comment = forms.CharField()
```

This form will include three default **TextInput** widgets, with default rendering – no CSS class, no extra attributes. This means that the input boxes provided for each widget will be rendered exactly the same:

```
>>> f = CommentForm(auto_id=False)
>>> f.as_table()
<tr><th>Name:</th><td><input type="text" name="name" required></td></tr>
<tr><th>Url:</th><td><input type="url" name="url" required></td></tr>
<tr><th>Comment:</th><td><input type="text" name="comment" required></td></tr>
```

On a real Web page, you probably don't want every widget to look the same. You might want a larger input element for the comment, and you might want the 'name' widget to have some special CSS class. It is also possible to specify the 'type' attribute to take advantage of the new HTML5 input types. To do this, you use the **Widget.attrs** argument when creating the widget:

```
class CommentForm(forms.Form):
    name = forms.CharField(widget=forms.TextInput(attrs={'class': 'special'}))
    url = forms.URLField()
    comment = forms.CharField(widget=forms.TextInput(attrs={'size': '40'}))
```

You can also modify a widget in the form definition:

```
class CommentForm(forms.Form):
    name = forms.CharField()
    url = forms.URLField()
    comment = forms.CharField()

    name.widget.attrs.update({'class': 'special'})
    comment.widget.attrs.update(size='40')
```

Or if the field isn't declared directly on the form (such as model form fields), you can use the **Form.fields** attribute:

```
class CommentForm(forms.ModelForm):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.fields['name'].widget.attrs.update({'class': 'special'})
        self.fields['comment'].widget.attrs.update(size='40')
```

Django will then include the extra attributes in the rendered output:

```
>>> f = CommentForm(auto_id=False)
>>> f.as_table()
<tr><th>Name:</th><td><input type="text" name="name" class="special" required></td></tr>
<tr><th>Url:</th><td><input type="url" name="url" required></td></tr>
<tr><th>Comment:</th><td><input type="text" name="comment" size="40" required></td></tr>
```

You can also set the HTML **id** using **attrs**. See **BoundField.id_for_label** for an example.

Styling widget classes

With widgets, it is possible to add assets (`css` and `javascript`) and more deeply customize their appearance and behavior.

In a nutshell, you will need to subclass the widget and either define a "Media" inner class or create a "media" property.

These methods involve somewhat advanced Python programming and are described in detail in the Form Assets topic guide.

Base widget classes

Base widget classes `Widget` and `MultiWidget` are subclassed by all the built-in widgets and may serve as a foundation for custom widgets.

`Widget`

***class* `Widget`(*attrs=None*)**

This abstract class cannot be rendered, but provides the basic attribute `attrs`. You may also implement or override the `render()` method on custom widgets.

`attrs`

A dictionary containing HTML attributes to be set on the rendered widget.

```
>>> from django import forms
>>> name = forms.TextInput(attrs={'size': 10, 'title': 'Your name'})
>>> name.render('name', 'A name')
'<input title="Your name" type="text" name="name" value="A name" size="10">'
```

If you assign a value of `True` or `False` to an attribute, it will be rendered as an HTML5 boolean attribute:

```
>>> name = forms.TextInput(attrs={'required': True})
>>> name.render('name', 'A name')
'<input name="name" type="text" value="A name" required>'
>>>
>>> name = forms.TextInput(attrs={'required': False})
>>> name.render('name', 'A name')
'<input name="name" type="text" value="A name">'
```

`supports_microseconds`

An attribute that defaults to `True`. If set to `False`, the microseconds part of `datetime` and `time` values will be set to `0`.

`format_value`(*value*)

Cleans and returns a value for use in the widget template. `value` isn't guaranteed to be valid input, therefore subclass implementations should program defensively.

`get_context`(*name*, *value*, *attrs*)

Returns a dictionary of values to use when rendering the widget template. By default, the dictionary contains a single key, `'widget'`, which is a dictionary representation of the widget containing the following keys:

- `'name'`: The name of the field from the `name` argument.
- `'is_hidden'`: A boolean indicating whether or not this widget is hidden.
- `'required'`: A boolean indicating whether or not the field for this widget is required.
- `'value'`: The value as returned by `format_value()`.
- `'attrs'`: HTML attributes to be set on the rendered widget. The combination of the `attrs` attribute and the `attrs` argument.
- `'template_name'`: The value of `self.template_name`.

`Widget` subclasses can provide custom context values by overriding this method.

`id_for_label(id_)`

Returns the HTML ID attribute of this widget for use by a `<label>`, given the ID of the field. Returns `None` if an ID isn't available.

This hook is necessary because some widgets have multiple HTML elements and, thus, multiple IDs. In that case, this method should return an ID value that corresponds to the first ID in the widget's tags.

`render(name, value, attrs=None, renderer=None)`

Renders a widget to HTML using the given renderer. If `renderer` is `None`, the renderer from the `FORM_RENDERER` setting is used.

`value_from_datadict(data, files, name)`

Given a dictionary of data and this widget's name, returns the value of this widget. `files` may contain data coming from `request.FILES`. Returns `None` if a value wasn't provided. Note also that `value_from_datadict` may be called more than once during handling of form data, so if you customize it and add expensive processing, you should implement some caching mechanism yourself.

`value_omitted_from_data(data, files, name)`

Given `data` and `files` dictionaries and this widget's name, returns whether or not there's data or files for the widget.

The method's result affects whether or not a field in a model form falls back to its default

Special cases are `CheckboxInput`, `CheckboxSelectMultiple`, and `SelectMultiple`, which always return `False` because an unchecked checkbox and unselected `<select multiple>` don't appear in the data of an HTML form submission, so it's unknown whether or not the user submitted a value.

`use_required_attribute(initial)`

Given a form field's `initial` value, returns whether or not the widget can be rendered with the `required` HTML attribute. Forms use this method along with `Field.required` and `Form.use_required_attribute` to determine whether or not to display the `required` attribute for each field.

By default, returns `False` for hidden widgets and `True` otherwise. Special cases are `ClearableFileInput`, which returns `False` when `initial` is set, and `CheckboxSelectMultiple`, which always returns `False` because browser validation would require all checkboxes to be checked instead of at least one.

Override this method in custom widgets that aren't compatible with browser validation. For example, a WSYSIWG text editor widget backed by a hidden `textarea` element may want to always return `False` to avoid browser validation on the hidden field.

`MultiWidget`

***class* `MultiWidget(widgets, attrs=None)`**

A widget that is composed of multiple widgets. `MultiWidget` works hand in hand with the `MultiValueField`.

`MultiWidget` has one required argument:

`widgets`

An iterable containing the widgets needed.

And one required method:

`decompress(value)`

This method takes a single "compressed" value from the field and returns a list of "decompressed" values. The input value can be assumed valid, but not necessarily non-empty.

This method **must be implemented** by the subclass, and since the value may be empty, the implementation must be defensive.

The rationale behind "decompression" is that it is necessary to "split" the combined value of the form field into the values for each widget.

An example of this is how `SplitDateTimeWidget` turns a `datetime` value into a list with date and time split into two separate values:

```
from django.forms import MultiWidget

class SplitDateTimeWidget(MultiWidget):

    # ...

    def decompress(self, value):
        if value:
            return [value.date(), value.time()]
        return [None, None]
```

Tip

Note that **MultiValueField** has a complementary method **compress()** with the opposite responsibility - to combine cleaned values of all member fields into one.

It provides some custom context:

get_context(*name*, *value*, *attrs*)

In addition to the **'widget'** key described in **Widget.get_context()**, **MultiValueWidget** adds a **widget['subwidgets']** key.

These can be looped over in the widget template:

```
{% for subwidget in widget.subwidgets %}
    {% include subwidget.template_name with widget=subwidget %}
{% endfor %}
```

Here's an example widget which subclasses **MultiWidget** to display a date with the day, month, and year in different select boxes. This widget is intended to be used with a **DateField** rather than a **MultiValueField**, thus we have implemented **value_from_datadict()**:

```
from datetime import date
from django import forms

class DateSelectorWidget(forms.MultiWidget):
    def __init__(self, attrs=None):
        days = [(day, day) for day in range(1, 32)]
        months = [(month, month) for month in range(1, 13)]
        years = [(year, year) for year in [2018, 2019, 2020]]
        widgets = [
            forms.Select(attrs=attrs, choices=days),
            forms.Select(attrs=attrs, choices=months),
            forms.Select(attrs=attrs, choices=years),
        ]
        super().__init__(widgets, attrs)

    def decompress(self, value):
        if isinstance(value, date):
            return [value.day, value.month, value.year]
        elif isinstance(value, str):
            year, month, day = value.split('-')
            return [day, month, year]
        return [None, None, None]

    def value_from_datadict(self, data, files, name):
        day, month, year = super().value_from_datadict(data, files, name)
        # DateField expects a single string that it can parse into a date.
        return '{}-{}-{}'.format(year, month, day)
```

The constructor creates several **Select** widgets in a list. The **super()** method uses this list to setup the widget.

The required method `decompress()` breaks up a `datetime.date` value into the day, month, and year values corresponding to each widget. If an invalid date was selected, such as the non-existent 30th February, the `DateField` passes this method a string instead, so that needs parsing. The final `return` handles when `value` is `None`, meaning we don't have any defaults for our subwidgets.

The default implementation of `value_from_datadict()` returns a list of values corresponding to each `Widget`. This is appropriate when using a `MultiWidget` with a `MultiValueField`. But since we want to use this widget with a `DateField`, which takes a single value, we have overridden this method. The implementation here combines the data from the subwidgets into a string in the format that `DateField` expects.

Built-in widgets

Django provides a representation of all the basic HTML widgets, plus some commonly used groups of widgets in the `django.forms.widgets` module, including the input of text, various checkboxes and selectors, uploading files, and handling of multi-valued input.

Widgets handling input of text

These widgets make use of the HTML elements `input` and `textarea`.

`TextInput`

class `TextInput`

- `input_type`: `'text'`
- `template_name`: `'django/forms/widgets/text.html'`
- Renders as: `<input type="text" ...>`

`NumberInput`

class `NumberInput`

- `input_type`: `'number'`
- `template_name`: `'django/forms/widgets/number.html'`
- Renders as: `<input type="number" ...>`

Beware that not all browsers support entering localized numbers in `number` input types. Django itself avoids using them for fields having their `localize` property set to `True`.

`EmailInput`

class `EmailInput`

- `input_type`: `'email'`
- `template_name`: `'django/forms/widgets/email.html'`
- Renders as: `<input type="email" ...>`

`URLInput`

class `URLInput`

- `input_type`: `'url'`
- `template_name`: `'django/forms/widgets/url.html'`
- Renders as: `<input type="url" ...>`

`PasswordInput`

class `PasswordInput`

- **input_type**: **'password'**
- **template_name**: **'django/forms/widgets/password.html'**
- Renders as: **<input type="password" ...>**

Takes one optional argument:

render_value

Determines whether the widget will have a value filled in when the form is re-displayed after a validation error (default is **False**).

HiddenInput

class **HiddenInput**

- **input_type**: **'hidden'**
- **template_name**: **'django/forms/widgets/hidden.html'**
- Renders as: **<input type="hidden" ...>**

Note that there also is a **MultipleHiddenInput** widget that encapsulates a set of hidden input elements.

DateInput

class **DateInput**

- **input_type**: **'text'**
- **template_name**: **'django/forms/widgets/date.html'**
- Renders as: **<input type="text" ...>**

Takes same arguments as **TextInput**, with one more optional argument:

format

The format in which this field's initial value will be displayed.

If no **format** argument is provided, the default format is the first format found in **DATE_INPUT_FORMATS** and respects Format localization.

DateTimeInput

class **DateTimeInput**

- **input_type**: **'text'**
- **template_name**: **'django/forms/widgets/datetime.html'**
- Renders as: **<input type="text" ...>**

Takes same arguments as **TextInput**, with one more optional argument:

format

The format in which this field's initial value will be displayed.

If no **format** argument is provided, the default format is the first format found in **DATETIME_INPUT_FORMATS** and respects Format localization.

By default, the microseconds part of the time value is always set to **0**. If microseconds are required, use a subclass with the **supports_microseconds** attribute set to **True**.

TimeInput

class **TimeInput**

- **input_type**: **'text'**
- **template_name**: **'django/forms/widgets/time.html'**

- Renders as: `<input type="text" ...>`

Takes same arguments as `TextInput`, with one more optional argument:

`format`

The format in which this field's initial value will be displayed.

If no `format` argument is provided, the default format is the first format found in `TIME_INPUT_FORMATS` and respects Format localization.

For the treatment of microseconds, see `DateTimeInput`.

`Textarea`

class `Textarea`

- `template_name`: `'django/forms/widgets/textarea.html'`
- Renders as: `<textarea>...</textarea>`

Selector and checkbox widgets

These widgets make use of the HTML elements `<select>`, `<input type="checkbox">`, and `<input type="radio">`.

Widgets that render multiple choices have an `option_template_name` attribute that specifies the template used to render each choice. For example, for the `Select` widget, `select_option.html` renders the `<option>` for a `<select>`.

`CheckboxInput`

class `CheckboxInput`

- `input_type`: `'checkbox'`
- `template_name`: `'django/forms/widgets/checkbox.html'`
- Renders as: `<input type="checkbox" ...>`

Takes one optional argument:

`check_test`

A callable that takes the value of the `CheckboxInput` and returns `True` if the checkbox should be checked for that value.

`Select`

class `Select`

- `template_name`: `'django/forms/widgets/select.html'`
- `option_template_name`: `'django/forms/widgets/select_option.html'`
- Renders as: `<select><option ...>...</select>`

`choices`

This attribute is optional when the form field does not have a `choices` attribute. If it does, it will override anything you set here when the attribute is updated on the `Field`.

`NullBooleanSelect`

class `NullBooleanSelect`

- `template_name`: `'django/forms/widgets/select.html'`
- `option_template_name`: `'django/forms/widgets/select_option.html'`

Select widget with options 'Unknown', 'Yes' and 'No'

SelectMultiple

class **SelectMultiple**

- **template_name**: **'django/forms/widgets/select.html'**
- **option_template_name**: **'django/forms/widgets/select_option.html'**

Similar to **Select**, but allows multiple selection: **<select multiple>...</select>**

RadioSelect

class **RadioSelect**

- **template_name**: **'django/forms/widgets/radio.html'**
- **option_template_name**: **'django/forms/widgets/radio_option.html'**

Similar to **Select**, but rendered as a list of radio buttons within **<li>** tags:

```
<ul>
  <li><input type="radio" name="..."></li>
  ...
</ul>
```

For more granular control over the generated markup, you can loop over the radio buttons in the template. Assuming a form **myform** with a field **beatles** that uses a **RadioSelect** as its widget:

```
{% for radio in myform.beatles %}
<div class="myradio">
    {{ radio }}
</div>
{% endfor %}
```

This would generate the following HTML:

```
<div class="myradio">
    <label for="id_beatles_0"><input id="id_beatles_0" name="beatles" type="radio" value="john" required>
John</label>
</div>
<div class="myradio">
    <label for="id_beatles_1"><input id="id_beatles_1" name="beatles" type="radio" value="paul" required>
Paul</label>
</div>
<div class="myradio">
    <label for="id_beatles_2"><input id="id_beatles_2" name="beatles" type="radio" value="george" required>
George</label>
</div>
<div class="myradio">
    <label for="id_beatles_3"><input id="id_beatles_3" name="beatles" type="radio" value="ringo" required>
Ringo</label>
</div>
```

That included the **<label>** tags. To get more granular, you can use each radio button's **tag**, **choice_label** and **id_for_label** attributes. For example, this template…

```
{% for radio in myform.beatles %}
    <label for="{{ radio.id_for_label }}">
        {{ radio.choice_label }}
        <span class="radio">{{ radio.tag }}</span>
    </label>
{% endfor %}
```

…will result in the following HTML:

```
<label for="id_beatles_0">
    John
    <span class="radio"><input id="id_beatles_0" name="beatles" type="radio" value="john" required></span>
</label>

<label for="id_beatles_1">
    Paul
    <span class="radio"><input id="id_beatles_1" name="beatles" type="radio" value="paul" required></span>
</label>

<label for="id_beatles_2">
    George
    <span class="radio"><input id="id_beatles_2" name="beatles" type="radio" value="george" required></span>
</label>

<label for="id_beatles_3">
    Ringo
    <span class="radio"><input id="id_beatles_3" name="beatles" type="radio" value="ringo" required></span>
</label>
```

If you decide not to loop over the radio buttons – e.g., if your template includes **`{{ myform.beatles }}`** – they'll be output in a **`<ul>`** with **`<li>`** tags, as above.

The outer **`<ul>`** container receives the **`id`** attribute of the widget, if defined, or **`BoundField.auto_id`** otherwise.

When looping over the radio buttons, the **`label`** and **`input`** tags include **`for`** and **`id`** attributes, respectively. Each radio button has an **`id_for_label`** attribute to output the element's ID.

`CheckboxSelectMultiple`

***class* `CheckboxSelectMultiple`**

- **`template_name`**: **`'django/forms/widgets/checkbox_select.html'`**
- **`option_template_name`**: **`'django/forms/widgets/checkbox_option.html'`**

Similar to **`SelectMultiple`**, but rendered as a list of checkboxes:

```
<ul>
  <li><input type="checkbox" name="..." ></li>
  ...
</ul>
```

The outer **`<ul>`** container receives the **`id`** attribute of the widget, if defined, or **`BoundField.auto_id`** otherwise.

Like **`RadioSelect`**, you can loop over the individual checkboxes for the widget's choices. Unlike **`RadioSelect`**, the checkboxes won't include the **`required`** HTML attribute if the field is required because browser validation would require all checkboxes to be checked instead of at least one.

When looping over the checkboxes, the **`label`** and **`input`** tags include **`for`** and **`id`** attributes, respectively. Each checkbox has an **`id_for_label`** attribute to output the element's ID.

File upload widgets

`FileInput`

class `FileInput`

- `template_name`: `'django/forms/widgets/file.html'`
- Renders as: `<input type="file" ...>`

`ClearableFileInput`

class `ClearableFileInput`

- `template_name`: `'django/forms/widgets/clearable_file_input.html'`
- Renders as: `<input type="file" ...>` with an additional checkbox input to clear the field's value, if the field is not required and has initial data.

Composite widgets

`MultipleHiddenInput`

class `MultipleHiddenInput`

- `template_name`: `'django/forms/widgets/multiple_hidden.html'`
- Renders as: multiple `<input type="hidden" ...>` tags

A widget that handles multiple hidden widgets for fields that have a list of values.

`SplitDateTimeWidget`

class `SplitDateTimeWidget`

- `template_name`: `'django/forms/widgets/splitdatetime.html'`

Wrapper (using `MultiWidget`) around two widgets: `DateInput` for the date, and `TimeInput` for the time. Must be used with `SplitDateTimeField` rather than `DateTimeField`.

`SplitDateTimeWidget` has several optional arguments:

`date_format`

Similar to `DateInput.format`

`time_format`

Similar to `TimeInput.format`

`date_attrs`

`time_attrs`

Similar to `Widget.attrs`. A dictionary containing HTML attributes to be set on the rendered `DateInput` and `TimeInput` widgets, respectively. If these attributes aren't set, `Widget.attrs` is used instead.

`SplitHiddenDateTimeWidget`

class `SplitHiddenDateTimeWidget`

- `template_name`: `'django/forms/widgets/splithiddendatetime.html'`

Similar to `SplitDateTimeWidget`, but uses `HiddenInput` for both date and time.

SelectDateWidget

class **SelectDateWidget**

- **template_name**: **'django/forms/widgets/select_date.html'**

Wrapper around three **Select** widgets: one each for month, day, and year.

Takes several optional arguments:

years

An optional list/tuple of years to use in the “year” select box. The default is a list containing the current year and the next 9 years.

months

An optional dict of months to use in the “months” select box.

The keys of the dict correspond to the month number (1-indexed) and the values are the displayed months:

```
MONTHS = {
    1:_('jan'), 2:_('feb'), 3:_('mar'), 4:_('apr'),
    5:_('may'), 6:_('jun'), 7:_('jul'), 8:_('aug'),
    9:_('sep'), 10:_('oct'), 11:_('nov'), 12:_('dec')
}
```

empty_label

If the **DateField** is not required, **SelectDateWidget** will have an empty choice at the top of the list (which is **---** by default). You can change the text of this label with the **empty_label** attribute. **empty_label** can be a **string**, **list**, or **tuple**. When a string is used, all select boxes will each have an empty choice with this label. If **empty_label** is a **list** or **tuple** of 3 string elements, the select boxes will have their own custom label. The labels should be in this order **('year_label', 'month_label', 'day_label')**.

```
# A custom empty label with string
field1 = forms.DateField(widget=SelectDateWidget(empty_label="Nothing"))

# A custom empty label with tuple
field1 = forms.DateField(
    widget=SelectDateWidget(
        empty_label=("Choose Year", "Choose Month", "Choose Day"),
    ),
)
```

‹ The form rendering API

Form and field validation ›

BACK TO TOP

Contents

Browse

- Prev: The form rendering API
- Next: Form and field validation
- Table of contents

- General Index
- Python Module Index

You are here:

- Django 3.0 documentation
  - API Reference
    - Forms
      - Widgets

Getting help

FAQ
Try the FAQ — it's got answers to many common questions.

Index, Module Index, or Table of Contents
Handy when looking for specific information.

django-users mailing list
Search for information in the archives of the django-users mailing list, or post a question.

#django IRC channel
Ask a question in the #django IRC channel, or search the IRC logs to see if it's been asked before.

Ticket tracker
Report bugs with Django or Django documentation in our ticket tracker.

Download:

Offline (Django 3.0): HTML | PDF | ePub
Provided by Read the Docs.

Report a Security Issue

Follow Us

GitHub

Twitter

News RSS

Django Users Mailing List

django

Hosting by

rackspace

Design by

threespot

&

andrevv

Getting Help

Language: **en**

Documentation version: **3.0**