

Django v1.2 documentation

[« previous](#) | [up](#) | [next »](#)

Unicode data

New in Django 1.0: [Please, see the release notes](#)

Django natively supports Unicode data everywhere. Providing your database can somehow store the data, you can safely pass around Unicode strings to templates, models and the database.

This document tells you what you need to know if you're writing applications that use data or templates that are encoded in something other than ASCII.

Creating the database

Make sure your database is configured to be able to store arbitrary string data. Normally, this means giving it an encoding of UTF-8 or UTF-16. If you use a more restrictive encoding – for example, latin1 (iso8859-1) – you won't be able to store certain characters in the database, and information will be lost.

- MySQL users, refer to the [MySQL manual](#) (section 9.1.3.2 for MySQL 5.1) for details on how to set or alter the database character set encoding.
- PostgreSQL users, refer to the [PostgreSQL manual](#) (section 21.2.2 in PostgreSQL 8) for details on creating databases with the correct encoding.
- SQLite users, there is nothing you need to do. SQLite always uses UTF-8 for internal encoding.

All of Django's database backends automatically convert Unicode strings into the appropriate encoding for talking to the database. They also automatically convert strings retrieved from the database into Python Unicode strings. You don't even need to tell Django what encoding your database uses: that is handled transparently.

For more, see the section "The database API" below.

General string handling

Whenever you use strings with Django – e.g., in database lookups, template rendering or anywhere else – you have two choices for encoding those strings. You can use Unicode strings, or you can use normal strings (sometimes called "bytestrings") that are encoded using UTF-8.

Warning

A bytestring does not carry any information with it about its encoding. For that reason, we have to make an assumption, and Django assumes that all bytestrings are in UTF-8.

If you pass a string to Django that has been encoded in some other format, things will go wrong in interesting ways. Usually, Django will raise a `UnicodeDecodeError` at some point.

If your code only uses ASCII data, it's safe to use your normal strings, passing them around at will, because ASCII is a subset of UTF-8.

Don't be fooled into thinking that if your `DEFAULT_CHARSET` setting is set to something other than `'utf-8'` you can use that other encoding in your bytestrings! `DEFAULT_CHARSET` only applies to the strings generated as the result of template rendering (and e-mail). Django will always assume UTF-8 encoding for internal bytestrings. The reason for this is that the `DEFAULT_CHARSET` setting is not actually under your control (if you are the application developer). It's under the control of the person installing and using your application – and if that person chooses a different setting, your code must still continue to work. Ergo, it cannot rely on that setting.

In most cases when Django is dealing with strings, it will convert them to Unicode strings before doing anything else. So, as a general rule, if you pass in a bytestring, be prepared to receive a Unicode string back in the result.

Translated strings

Aside from Unicode strings and bytestrings, there's a third type of string-like object you may encounter when using Django. The framework's internationalization features introduce the concept of a "lazy translation" – a string that has been marked as translated but whose actual translation result isn't determined until the object is used in a string. This feature is useful in cases where the translation locale is unknown until the string is used, even though the string might have originally been created when the code was first imported.

Normally, you won't have to worry about lazy translations. Just be aware that if you examine an object and it claims to be a `django.utils.functional.__proxy__` object, it is a lazy translation. Calling `unicode()`

Table Of Contents

- Unicode data
 - Creating the database
 - General string handling
 - Translated strings
 - Useful utility functions
 - Conversion functions
 - URI and IRI handling
- Models
 - Choosing between `__str__()` and `__unicode__()`
 - Taking care in `get_absolute_url()`
- The database API
- Templates
 - Template tags and filters
- E-mail
- Form submission

Browse

- Prev: [The Django template language: For Python programmers](#)
- Next: [Django Utils](#)

You are here:

- [Django v1.2 documentation](#)
 - [API Reference](#)
 - [Unicode data](#)

This Page

- [Show Source](#)

Quick search

Enter search terms or a module, class or function name.

Last update:

Jul 05, 2010

with the lazy translation as the argument will generate a Unicode string in the current locale.

For more details about lazy translation objects, refer to the [internationalization](#) documentation.

Useful utility functions

Because some string operations come up again and again, Django ships with a few useful functions that should make working with Unicode and bytestring objects a bit easier.

Conversion functions

The `django.utils.encoding` module contains a few functions that are handy for converting back and forth between Unicode and bytestrings.

- `smart_unicode(s, encoding='utf-8', strings_only=False, errors='strict')` converts its input to a Unicode string. The `encoding` parameter specifies the input encoding. (For example, Django uses this internally when processing form input data, which might not be UTF-8 encoded.) The `strings_only` parameter, if set to `True`, will result in Python numbers, booleans and `None` not being converted to a string (they keep their original types). The `errors` parameter takes any of the values that are accepted by Python's `unicode()` function for its error handling.

If you pass `smart_unicode()` an object that has a `__unicode__` method, it will use that method to do the conversion.

- `force_unicode(s, encoding='utf-8', strings_only=False, errors='strict')` is identical to `smart_unicode()` in almost all cases. The difference is when the first argument is a [lazy translation](#) instance. While `smart_unicode()` preserves lazy translations, `force_unicode()` forces those objects to a Unicode string (causing the translation to occur). Normally, you'll want to use `smart_unicode()`. However, `force_unicode()` is useful in template tags and filters that absolutely must have a string to work with, not just something that can be converted to a string.
- `smart_str(s, encoding='utf-8', strings_only=False, errors='strict')` is essentially the opposite of `smart_unicode()`. It forces the first argument to a bytestring. The `strings_only` parameter has the same behavior as for `smart_unicode()` and `force_unicode()`. This is slightly different semantics from Python's builtin `str()` function, but the difference is needed in a few places within Django's internals.

Normally, you'll only need to use `smart_unicode()`. Call it as early as possible on any input data that might be either Unicode or a bytestring, and from then on, you can treat the result as always being Unicode.

URI and IRI handling

Web frameworks have to deal with URLs (which are a type of [IRI](#)). One requirement of URLs is that they are encoded using only ASCII characters. However, in an international environment, you might need to construct a URL from an [IRI](#) – very loosely speaking, a URI that can contain Unicode characters. Quoting and converting an IRI to URI can be a little tricky, so Django provides some assistance.

- The function `django.utils.encoding.iri_to_uri()` implements the conversion from IRI to URI as required by the specification ([RFC 3987](#)).
- The functions `django.utils.http.urlquote()` and `django.utils.http.urlquote_plus()` are versions of Python's standard `urllib.quote()` and `urllib.quote_plus()` that work with non-ASCII characters. (The data is converted to UTF-8 prior to encoding.)

These two groups of functions have slightly different purposes, and it's important to keep them straight. Normally, you would use `urlquote()` on the individual portions of the IRI or URI path so that any reserved characters such as `'&'` or `'%'` are correctly encoded. Then, you apply `iri_to_uri()` to the full IRI and it converts any non-ASCII characters to the correct encoded values.

Note

Technically, it isn't correct to say that `iri_to_uri()` implements the full algorithm in the IRI specification. It doesn't (yet) perform the international domain name encoding portion of the algorithm.

The `iri_to_uri()` function will not change ASCII characters that are otherwise permitted in a URL. So, for example, the character `'%'` is not further encoded when passed to `iri_to_uri()`. This means you can pass a full URL to this function and it will not mess up the query string or anything like that.

An example might clarify things here:

```
>>> urlquote(u'Paris & Orléans')
u'Paris%20%26%200r1%C3%A9ans'
>>> iri_to_uri(u'/favorites/François/%s' % urlquote(u'Paris & Orléans'))
'/favorites/Fran%C3%A7ois/Paris%20%26%200r1%C3%A9ans'
```

If you look carefully, you can see that the portion that was generated by `urlquote()` in the second example was not double-quoted when passed to `iri_to_uri()`. This is a very important and useful feature. It means that you can construct your IRI without worrying about whether it contains non-ASCII characters and then, right at the end, call `iri_to_uri()` on the result.

The `iri_to_uri()` function is also idempotent, which means the following is always true:

```
iri_to_uri(iri_to_uri(some_string)) = iri_to_uri(some_string)
```

So you can safely call it multiple times on the same IRI without risking double-quoting problems.

Models

Because all strings are returned from the database as Unicode strings, model fields that are character based (`CharField`, `TextField`, `URLField`, etc) will contain Unicode values when Django retrieves data from the database. This is always the case, even if the data could fit into an ASCII bytestring.

You can pass in bytestrings when creating a model or populating a field, and Django will convert it to Unicode when it needs to.

Choosing between `__str__()` and `__unicode__()`

One consequence of using Unicode by default is that you have to take some care when printing data from the model.

In particular, rather than giving your model a `__str__()` method, we recommended you implement a `__unicode__()` method. In the `__unicode__()` method, you can quite safely return the values of all your fields without having to worry about whether they fit into a bytestring or not. (The way Python works, the result of `__str__()` is always a bytestring, even if you accidentally try to return a Unicode object).

You can still create a `__str__()` method on your models if you want, of course, but you shouldn't need to do this unless you have a good reason. Django's `Model` base class automatically provides a `__str__()` implementation that calls `__unicode__()` and encodes the result into UTF-8. This means you'll normally only need to implement a `__unicode__()` method and let Django handle the coercion to a bytestring when required.

Taking care in `get_absolute_url()`

URLs can only contain ASCII characters. If you're constructing a URL from pieces of data that might be non-ASCII, be careful to encode the results in a way that is suitable for a URL. The `django.db.models permalink()` decorator handles this for you automatically.

If you're constructing a URL manually (i.e., not using the `permalink()` decorator), you'll need to take care of the encoding yourself. In this case, use the `iri_to_uri()` and `urlquote()` functions that were documented [above](#). For example:

```
from django.utils.encoding import iri_to_uri
from django.utils.http import urlquote

def get_absolute_url(self):
    url = u'/person/%s/?x=0&y=0' % urlquote(self.location)
    return iri_to_uri(url)
```

This function returns a correctly encoded URL even if `self.location` is something like "Jack visited Paris & Orléans". (In fact, the `iri_to_uri()` call isn't strictly necessary in the above example, because all the non-ASCII characters would have been removed in quoting in the first line.)

The database API

You can pass either Unicode strings or UTF-8 bytestrings as arguments to `filter()` methods and the like in the database API. The following two queriesets are identical:

```
qs = People.objects.filter(name__contains=u'Ã')
qs = People.objects.filter(name__contains='\xc3\x85') # UTF-8 encoding of Ã
```

Templates

You can use either Unicode or bytestrings when creating templates manually:

```
from django.template import Template
t1 = Template('This is a bytestring template.')
t2 = Template(u'This is a Unicode template.')
```

But the common case is to read templates from the filesystem, and this creates a slight complication: not all filesystems store their data encoded as UTF-8. If your template files are not stored with a UTF-8 encoding, set the `FILE_CHARSET` setting to the encoding of the files on disk. When Django reads in a

template file, it will convert the data from this encoding to Unicode. (`FILE_CHARSET` is set to 'utf-8' by default.)

The `DEFAULT_CHARSET` setting controls the encoding of rendered templates. This is set to UTF-8 by default.

Template tags and filters

A couple of tips to remember when writing your own template tags and filters:

- Always return Unicode strings from a template tag's `render()` method and from template filters.
- Use `force_unicode()` in preference to `smart_unicode()` in these places. Tag rendering and filter calls occur as the template is being rendered, so there is no advantage to postponing the conversion of lazy translation objects into strings. It's easier to work solely with Unicode strings at that point.

E-mail

Django's e-mail framework (in `django.core.mail`) supports Unicode transparently. You can use Unicode data in the message bodies and any headers. However, you're still obligated to respect the requirements of the e-mail specifications, so, for example, e-mail addresses should use only ASCII characters.

The following code example demonstrates that everything except e-mail addresses can be non-ASCII:

```
from django.core.mail import EmailMessage

subject = u'My visit to Sør-Trøndelag'
sender = u'Arnbjörg Ráðormsdóttir <arnbjorg@example.com>'
recipients = ['Fred <fred@example.com>']
body = u'...'
EmailMessage(subject, body, sender, recipients).send()
```

Form submission

HTML form submission is a tricky area. There's no guarantee that the submission will include encoding information, which means the framework might have to guess at the encoding of submitted data.

Django adopts a "lazy" approach to decoding form data. The data in an `HttpRequest` object is only decoded when you access it. In fact, most of the data is not decoded at all. Only the `HttpRequest.GET` and `HttpRequest.POST` data structures have any decoding applied to them. Those two fields will return their members as Unicode data. All other attributes and methods of `HttpRequest` return data exactly as it was submitted by the client.

By default, the `DEFAULT_CHARSET` setting is used as the assumed encoding for form data. If you need to change this for a particular form, you can set the `encoding` attribute on an `HttpRequest` instance. For example:

```
def some_view(request):
    # We know that the data must be encoded as KOI8-R (for some reason).
    request.encoding = 'koi8-r'
    ...
```

You can even change the encoding after having accessed `request.GET` or `request.POST`, and all subsequent accesses will use the new encoding.

Most developers won't need to worry about changing form encoding, but this is a useful feature for applications that talk to legacy systems whose encoding you cannot control.

Django does not decode the data of file uploads, because that data is normally treated as collections of bytes, rather than strings. Any automatic decoding there would alter the meaning of the stream of bytes.

[« previous](#) | [up](#) | [next »](#)