Babel Documentation

Release 2.2

The Babel Team

February 03, 2016

Contents

Babel is an integrated collection of utilities that assist in internationalizing and localizing Python applications, with an emphasis on web-based applications.

CHAPTER 1

User Documentation

The user documentation explains some core concept of the library and gives some information about how it can be used.

1.1 Introduction

The functionality Babel provides for internationalization (I18n) and localization (L10N) can be separated into two different aspects:

- tools to build and work with `gettext` message catalogs, and
- a Python interface to the CLDR (Common Locale Data Repository), providing access to various locale display names, localized number and date formatting, etc.

1.1.1 Message Catalogs

While the Python standard library includes a `gettext` module that enables applications to use message catalogs, it requires developers to build these catalogs using GNU tools such as `xgettext`, `msgmerge`, and `msgfmt`. And while `xgettext` does have support for extracting messages from Python files, it does not know how to deal with other kinds of files commonly found in Python web-applications, such as templates, nor does it provide an easy extensibility mechanism to add such support.

Babel addresses this by providing a framework where various extraction methods can be plugged in to a larger message extraction framework, and also removes the dependency on the GNU `gettext` tools for common tasks, as these aren't necessarily available on all platforms. See *Working with Message Catalogs* for details on this aspect of Babel.

1.1.2 Locale Data

Furthermore, while the Python standard library does include support for basic localization with respect to the formatting of numbers and dates (the `locale` module, among others), this support is based on the assumption that there will be only one specific locale used per process (at least simultaneously.) Also, it doesn't provide access to other kinds of locale data, such as the localized names of countries, languages, or time-zones, which are frequently needed in web-based applications.

For these requirements, Babel includes data extracted from the Common Locale Data Repository (CLDR), and provides a number of convenient methods for accessing and using this data. See *Locale Data*, *Date and Time*, and *Number Formatting* for more information on this aspect of Babel.

1.2 Installation

Babel is distributed as a standard Python package fully set up with all the dependencies it needs. It primarily depends on the excellent pytz library for timezone handling. To install it you can use `easy_install` or `pip`.

1.2.1 virtualenv

Virtualenv is probably what you want to use during development, and if you have shell access to your production machines, you'll probably want to use it there, too.

If you are on Mac OS X or Linux, chances are that one of the following two commands will work for you:

```
$ sudo easy_install virtualenv
```

If you are on Windows and don't have the *easy_install* command, you must install it first. Check the *pip and distribute on Windows* section for more information about how to do that. Once you have it installed, run the same commands as above, but without the *sudo* prefix.

Once you have virtualenv installed, just fire up a shell and create your own environment. I usually create a project folder and a *venv* folder within:

```
$ mkdir myproject
$ cd myproject
$ virtualenv venv
New python executable in venv/bin/python
Installing distribute.............done.
```

Now, whenever you want to work on a project, you only have to activate the corresponding environment. On OS X and Linux, do the following:

```
$ . venv/bin/activate
```

If you are a Windows user, the following command is for you:

```
$ venv\scripts\activate
```

Either way, you should now be using your virtualenv (notice how the prompt of your shell has changed to show the active environment).

Now you can just enter the following command to get Babel installed in your virtualenv:

```
$ pip install Babel
```

A few seconds later and you are good to go.

1.2.2 System-Wide Installation

This is possible as well, though I do not recommend it. Just run *pip* with root privileges:

```
$ sudo pip install Babel
```

If *pip* is not available on your system you can use *easy_install*.

(On Windows systems, run it in a command-prompt window with administrator privileges, and leave out *sudo*.)

1.2.3 Living on the Edge

If you want to work with the latest version of Babel, you will need to use a git checkout.

Get the git checkout in a new virtualenv and run in development mode:

```
$ git clone http://github.com/python-babel/babel.git
Initialized empty Git repository in ~/dev/babel/.git/
$ cd babel
$ virtualenv venv
New python executable in venv/bin/python
Installing distribute.............done.
$ . venv/bin/activate
$ python setup.py import_cldr
$ pip install --editable .
...
Finished processing dependencies for Babel
```

Make sure to not forget about the `import_cldr` step because otherwise you will be missing the locale data. This custom command will download the most appropriate CLDR release from the official website and convert it for Babel.

This will pull also in the dependencies and activate the git head as the current version inside the virtualenv. Then all you have to do is run `git pull origin` to update to the latest version. If the CLDR data changes you will have to re-run `python setup.py import_cldr`.

1.2.4 *pip* and *distribute* on Windows

On Windows, installation of *easy_install* is a little bit trickier, but still quite easy. The easiest way to do it is to download the distribute_setup.py file and run it. The easiest way to run the file is to open your downloads folder and double-click on the file.

Next, add the *easy_install* command and other Python scripts to the command search path, by adding your Python installation's Scripts folder to the *PATH* environment variable. To do that, right-click on the "Computer" icon on the Desktop or in the Start menu, and choose "Properties". Then click on "Advanced System settings" (in Windows XP, click on the "Advanced" tab instead). Then click on the "Environment variables" button. Finally, double-click on the "Path" variable in the "System variables" section, and add the path of your Python interpreter's Scripts folder. Be sure to delimit it from existing values with a semicolon. Assuming you are using Python 2.7 on the default path, add the following value:

```
;C:\Python27\Scripts
```

And you are done! To check that it worked, open the Command Prompt and execute `easy_install`. If you have User Account Control enabled on Windows Vista or Windows 7, it should prompt you for administrator privileges.

Now that you have `easy_install`, you can use it to install `pip`:

```
> easy_install pip
```

1.3 Locale Data

While *message catalogs* allow you to localize any messages in your application, there are a number of strings that are used in many applications for which translations are readily available.

Imagine for example you have a list of countries that users can choose from, and you'd like to display the names of those countries in the language the user prefers. Instead of translating all those country names yourself in your application, you can make use of the translations provided by the locale data included with Babel, which is based on the Common Locale Data Repository (CLDR) developed and maintained by the Unicode Consortium.

1.3.1 The `Locale` Class

You normally access such locale data through the *`Locale`* class provided by Babel:

```
>>> from babel import Locale
>>> locale = Locale('en', 'US')
>>> locale.territories['US']
u'United States'
>>> locale = Locale('es', 'MX')
>>> locale.territories['US']
u'Estados Unidos'
```

In addition to country/territory names, the locale data also provides access to names of languages, scripts, variants, time zones, and more. Some of the data is closely related to number and date formatting.

Most of the corresponding `Locale` properties return dictionaries, where the key is a code such as the ISO country and language codes. Consult the API documentation for references to the relevant specifications.

1.3.2 Likely Subtags

When dealing with locales you can run into the situation where a locale tag is not fully descriptive. For instance people commonly refer to `zh_TW` but that identifier does not resolve to a locale that the CLDR covers. Babel's locale identifier parser in that case will attempt to resolve the most likely subtag to end up with the intended locale:

```
>>> from babel import Locale
>>> Locale.parse('zh_TW')
Locale('zh', territory='TW', script='Hant')
```

This can also be used to find the most appropriate locale for a territory. In that case the territory code needs to be prefixed with `und` (unknown language identifier):

```
>>> Locale.parse('und_AZ')
Locale('az', territory='AZ', script='Latn')
>>> Locale.parse('und_DE')
Locale('de', territory='DE')
```

Babel currently cannot deal with fuzzy locales (a locale not fully backed by data files) so we only accept locales that are fully backed by CLDR data. This will change in the future, but for the time being this restriction is in place.

1.3.3 Locale Display Names

Locales itself can be used to describe the locale itself or other locales. This mainly means that given a locale object you can ask it for its canonical display name, the name of the language and other things. Since the locales cross-reference each other you can ask for locale names in any language supported by the CLDR:

```
>>> l = Locale.parse('de_DE')
>>> l.get_display_name('en_US')
u'German (Germany)'
>>> l.get_display_name('fr_FR')
u'allemand (Allemagne)'
```

Display names include all the information to uniquely identify a locale (language, territory, script and variant) which is often not what you want. You can also ask for the information in parts:

```
>>> l.get_language_name('de_DE')
u'Deutsch'
>>> l.get_language_name('it_IT')
u'tedesco'
>>> l.get_territory_name('it_IT')
u'Germania'
>>> l.get_territory_name('pt_PT')
u'Alemanha'
```

1.3.4 Calendar Display Names

The *Locale* class provides access to many locale display names related to calendar display, such as the names of weekdays or months.

These display names are of course used for date formatting, but can also be used, for example, to show a list of months to the user in their preferred language:

```
>>> locale = Locale('es')
>>> month_names = locale.months['format']['wide'].items()
>>> for idx, name in sorted(month_names):
...     print name
enero
febrero
marzo
abril
mayo
junio
julio
agosto
septiembre
octubre
noviembre
diciembre
```

1.4 Date and Time

When working with date and time information in Python, you commonly use the classes `date`, `datetime` and/or `time` from the *datetime* package. Babel provides functions for locale-specific formatting of those objects in its `dates` module:

```
>>> from datetime import date, datetime, time
>>> from babel.dates import format_date, format_datetime, format_time

>>> d = date(2007, 4, 1)
>>> format_date(d, locale='en')
u'Apr 1, 2007'
```

```
>>> format_date(d, locale='de_DE')
u'01.04.2007'
```

As this example demonstrates, Babel will automatically choose a date format that is appropriate for the requested locale.

The `format_*()` functions also accept an optional `format` argument, which allows you to choose between one of four format variations:

- `short`,
- `medium` (the default),
- `long`, and
- `full`.

For example:

```
>>> format_date(d, format='short', locale='en')
u'4/1/07'
>>> format_date(d, format='long', locale='en')
u'April 1, 2007'
>>> format_date(d, format='full', locale='en')
u'Sunday, April 1, 2007'
```

1.4.1 Core Time Concepts

Working with dates and time can be a complicated thing. Babel attempts to simplify working with them by making some decisions for you. Python's datetime module has different ways to deal with times and dates: naive and timezone-aware datetime objects.

Babel generally recommends you to store all your time in naive datetime objects and treat them as UTC at all times. This simplifies dealing with time a lot because otherwise you can get into the hairy situation where you are dealing with datetime objects of different timezones. That is tricky because there are situations where time can be ambiguous. This is usually the case when dealing with dates around timezone transitions. The most common case of timezone transition is changes between daylight saving time and standard time.

As such we recommend to always use UTC internally and only reformat to local time when returning dates to users. At that point the timezone the user has selected can usually be established and Babel can automatically rebase the time for you.

To get the current time use the `utcnow()` method of the `datetime` object. It will return a naive `datetime` object in UTC.

For more information about timezones see *Time-zone Support*.

1.4.2 Pattern Syntax

While Babel makes it simple to use the appropriate date/time format for a given locale, you can also force it to use custom patterns. Note that Babel uses different patterns for specifying number and date formats compared to the Python equivalents (such as `time.strftime()`), which have mostly been inherited from C and POSIX. The patterns used in Babel are based on the Locale Data Markup Language specification (LDML), which defines them as follows:

> A date/time pattern is a string of characters, where specific strings of characters are replaced with date and time data from a calendar when formatting or used to generate data for a calendar when parsing. [...]

> Characters may be used multiple times. For example, if `y` is used for the year, `yy` might produce "99", whereas `yyyy` produces "1999". For most numerical fields, the number of characters specifies the field width. For example, if `h` is the hour, `h` might produce "5", but `hh` produces "05". For some characters, the count specifies whether an abbreviated or full form should be used [...]
>
> Two single quotes represent a literal single quote, either inside or outside single quotes. Text within single quotes is not interpreted in any way (except for two adjacent single quotes).

For example:

```
>>> d = date(2007, 4, 1)
>>> format_date(d, "EEE, MMM d, ''yy", locale='en')
u"Sun, Apr 1, '07"
>>> format_date(d, "EEEE, d.M.yyyy", locale='de')
u'Sonntag, 1.4.2007'

>>> t = time(15, 30)
>>> format_time(t, "hh 'o''clock' a", locale='en')
u"03 o'clock PM"
>>> format_time(t, 'H:mm a', locale='de')
u'15:30 nachm.'

>>> dt = datetime(2007, 4, 1, 15, 30)
>>> format_datetime(dt, "yyyyy.MMMM.dd GGG hh:mm a", locale='en')
u'02007.April.01 AD 03:30 PM'
```

The syntax for custom datetime format patterns is described in detail in the the Locale Data Markup Language specification. The following table is just a relatively brief overview.

Date Fields

Field	Symbol	Description
Era	G	Replaced with the era string for the current date. One to three letters for the abbreviated form, four lettersfor the long form, five for the narrow form
Year	y	Replaced by the year. Normally the length specifies the padding, but for two letters it also specifies the maximum length.
	Y	Same as y but uses the ISO year-week calendar.
	u	??
Quarter	Q	Use one or two for the numerical quarter, three for the abbreviation, or four for the full name.
	q	Use one or two for the numerical quarter, three for the abbreviation, or four for the full name.
Month	M	Use one or two for the numerical month, three for the abbreviation, or four for the full name, or five for the narrow name.
	L	Use one or two for the numerical month, three for the abbreviation, or four for the full name, or 5 for the narrow name.
Week	w	Week of year.
	W	Week of month.
Day	d	Day of month.
	D	Day of year.
	F	Day of week in month.
	g	??
Week day	E	Day of week. Use one through three letters for the short day, or four for the full name, or five for the narrow name.
	e	Local day of week. Same as E except adds a numeric value that will depend on the local starting day of the week, using one or two letters.
	c	??

Time Fields

Field	Symbol	Description
Period	a	AM or PM
Hour	h	Hour [1-12].
	H	Hour [0-23].
	K	Hour [0-11].
	k	Hour [1-24].
Minute	m	Use one or two for zero places padding.
Second	s	Use one or two for zero places padding.
	S	Fractional second, rounds to the count of letters.
	A	Milliseconds in day.
Timezone	z	Use one to three letters for the short timezone or four for the full name.
	Z	Use one to three letters for RFC 822, four letters for GMT format.
	v	Use one letter for short wall (generic) time, four for long wall time.
	V	Same as z, except that timezone abbreviations should be used regardless of whether they are in common use by the locale.

1.4.3 Time Delta Formatting

In addition to providing functions for formatting localized dates and times, the `babel.dates` module also provides a function to format the difference between two times, called a ''time delta''. These are usually represented as `datetime.timedelta` objects in Python, and it's also what you get when you subtract one `datetime` object from an other.

The `format_timedelta` function takes a `timedelta` object and returns a human-readable representation. This happens at the cost of precision, as it chooses only the most significant unit (such as year, week, or hour) of the difference, and displays that:

```
>>> from datetime import timedelta
>>> from babel.dates import format_timedelta
>>> delta = timedelta(days=6)
>>> format_timedelta(delta, locale='en_US')
u'1 week'
```

The resulting strings are based from the CLDR data, and are properly pluralized depending on the plural rules of the locale and the calculated number of units.

The function provides parameters for you to influence how this most significant unit is chosen: with `threshold` you set the value after which the presentation switches to the next larger unit, and with `granularity` you can limit the smallest unit to display:

```
>>> delta = timedelta(days=6)
>>> format_timedelta(delta, threshold=1.2, locale='en_US')
u'6 days'
>>> format_timedelta(delta, granularity='month', locale='en_US')
u'1 month'
```

1.4.4 Time-zone Support

Many of the verbose time formats include the time-zone, but time-zone information is not by default available for the Python `datetime` and `time` objects. The standard library includes only the abstract `tzinfo` class, which you need appropriate implementations for to actually use in your application. Babel includes a `tzinfo` implementation for UTC (Universal Time).

Babel uses pytz for real timezone support which includes the definitions of practically all of the time-zones used on the world, as well as important functions for reliably converting from UTC to local time, and vice versa. The module is generally wrapped for you so you can directly interface with it from within Babel:

```
>>> from datetime import time
>>> from babel.dates import get_timezone, UTC
>>> dt = datetime(2007, 04, 01, 15, 30, tzinfo=UTC)
>>> eastern = get_timezone('US/Eastern')
>>> format_datetime(dt, 'H:mm Z', tzinfo=eastern, locale='en_US')
u'11:30 -0400'
```

The recommended approach to deal with different time-zones in a Python application is to always use UTC internally, and only convert from/to the users time-zone when accepting user input and displaying date/time data, respectively. You can use Babel together with `pytz` to apply a time-zone to any `datetime` or `time` object for display, leaving the original information unchanged:

```
>>> british = get_timezone('Europe/London')
>>> format_datetime(dt, 'H:mm zzzz', tzinfo=british, locale='en_US')
u'16:30 British Summer Time'
```

Here, the given UTC time is adjusted to the "Europe/London" time-zone, and daylight savings time is taken into account. Daylight savings time is also applied to `format_time`, but because the actual date is unknown in that case, the current day is assumed to determine whether DST or standard time should be used.

For many timezones it's also possible to ask for the next timezone transition. This for instance is useful to answer the question "when do I have to move the clock forward next":

```
>>> t = get_next_timezone_transition('Europe/Vienna', datetime(2011, 3, 2))
>>> t
<TimezoneTransition CET -> CEST (2011-03-27 01:00:00)>
>>> t.from_offset
3600.0
>>> t.to_offset
7200.0
>>> t.from_tz
'CET'
>>> t.to_tz
'CEST'
```

Lastly Babel also provides support for working with the local timezone of your operating system. It's provided through the `LOCALTZ` constant:

```
>>> from babel.dates import LOCALTZ, get_timezone_name
>>> LOCALTZ
<DstTzInfo 'Europe/Vienna' CET+1:00:00 STD>
>>> get_timezone_name(LOCALTZ)
u'Central European Time'
```

Localized Time-zone Names

While the `Locale` class provides access to various locale display names related to time-zones, the process of building a localized name of a time-zone is actually quite complicated. Babel implements it in separately usable functions in the `babel.dates` module, most importantly the `get_timezone_name` function:

```
>>> from babel import Locale
>>> from babel.dates import get_timezone_name, get_timezone

>>> tz = get_timezone('Europe/Berlin')
>>> get_timezone_name(tz, locale=Locale.parse('pt_PT'))
u'Hora da Europa Central'
```

You can pass the function either a `datetime.tzinfo` object, or a `datetime.date` or `datetime.datetime` object. If you pass an actual date, the function will be able to take daylight savings time into account. If you pass just the time-zone, Babel does not know whether daylight savings time is in effect, so it uses a generic representation, which is useful for example to display a list of time-zones to the user.

```
>>> from datetime import datetime

>>> dt = tz.localize(datetime(2007, 8, 15))
>>> get_timezone_name(dt, locale=Locale.parse('de_DE'))
u'Mitteleurop\xe4ische Sommerzeit'
>>> get_timezone_name(tz, locale=Locale.parse('de_DE'))
u'Mitteleurop\xe4ische Zeit'
```

1.5 Number Formatting

Support for locale-specific formatting and parsing of numbers is provided by the `babel.numbers` module:

```
>>> from babel.numbers import format_number, format_decimal, format_percent
```

Examples:

```
>>> format_decimal(1.2345, locale='en_US')
u'1.234'
>>> format_decimal(1.2345, locale='sv_SE')
u'1,234'
>>> format_decimal(12345, locale='de_DE')
u'12.345'
```

1.5.1 Pattern Syntax

While Babel makes it simple to use the appropriate number format for a given locale, you can also force it to use custom patterns. As with date/time formatting patterns, the patterns Babel supports for number formatting are based on the Locale Data Markup Language specification (LDML).

Examples:

```
>>> format_decimal(-1.2345, format='#,##0.##;-#', locale='en')
u'-1.23'
>>> format_decimal(-1.2345, format='#,##0.##;(#)', locale='en')
u'(1.23)'
```

The syntax for custom number format patterns is described in detail in the the specification. The following table is just a relatively brief overview.

Symbol	Description
`0`	Digit
`1-9`	'1' through '9' indicate rounding.
`@`	Significant digit
`#`	Digit, zero shows as absent
`.`	Decimal separator or monetary decimal separator
`-`	Minus sign
`,`	Grouping separator
`E`	Separates mantissa and exponent in scientific notation
`+`	Prefix positive exponents with localized plus sign
`;`	Separates positive and negative subpatterns
`%`	Multiply by 100 and show as percentage
`‰`	Multiply by 1000 and show as per mille
`¤`	Currency sign, replaced by currency symbol. If doubled, replaced by international currency symbol. If tripled, uses the long form of the decimal symbol.
`'`	Used to quote special characters in a prefix or suffix
`*`	Pad escape, precedes pad character

1.5.2 Parsing Numbers

Babel can also parse numeric data in a locale-sensitive manner:

```
>>> from babel.numbers import parse_decimal, parse_number
```

Examples:

```
>>> parse_decimal('1,099.98', locale='en_US')
1099.98
>>> parse_decimal('1.099,98', locale='de')
1099.98
>>> parse_decimal('2,109,998', locale='de')
Traceback (most recent call last):
  ...
NumberFormatError: '2,109,998' is not a valid decimal number
```

Note: Number parsing is not properly implemented yet

1.6 Working with Message Catalogs

1.6.1 Introduction

The `gettext` translation system enables you to mark any strings used in your application as subject to localization, by wrapping them in functions such as `gettext(str)` and `ngettext(singular, plural, num)`. For brevity, the `gettext` function is often aliased to `_(str)`, so you can write:

```
print _("Hello")
```

instead of just:

```
print "Hello"
```

to make the string "Hello" localizable.

Message catalogs are collections of translations for such localizable messages used in an application. They are commonly stored in PO (Portable Object) and MO (Machine Object) files, the formats of which are defined by the GNU gettext tools and the GNU translation project.

The general procedure for building message catalogs looks something like this:

- use a tool (such as `xgettext`) to extract localizable strings from the code base and write them to a POT (PO Template) file.
- make a copy of the POT file for a specific locale (for example, "en_US") and start translating the messages
- use a tool such as `msgfmt` to compile the locale PO file into an binary MO file
- later, when code changes make it necessary to update the translations, you regenerate the POT file and merge the changes into the various locale-specific PO files, for example using `msgmerge`

Python provides the `gettext` module as part of the standard library, which enables applications to work with appropriately generated MO files.

As `gettext` provides a solid and well supported foundation for translating application messages, Babel does not reinvent the wheel, but rather reuses this infrastructure, and makes it easier to build message catalogs for Python applications.

1.6.2 Message Extraction

Babel provides functionality similar to that of the `xgettext` program, except that only extraction from Python source files is built-in, while support for other file formats can be added using a simple extension mechanism.

Unlike `xgettext`, which is usually invoked once for every file, the routines for message extraction in Babel operate on directories. While the per-file approach of `xgettext` works nicely with projects using a `Makefile`, Python projects rarely use `make`, and thus a different mechanism is needed for extracting messages from the heterogeneous collection of source files that many Python projects are composed of.

When message extraction is based on directories instead of individual files, there needs to be a way to configure which files should be treated in which manner. For example, while many projects may contain `.html` files, some of those files may be static HTML files that don't contain localizable message, while others may be Jinja2 templates, and still others may contain Genshi markup templates. Some projects may even mix HTML files for different templates languages (for whatever reason). Therefore the way in which messages are extracted from source files can not only depend on the file extension, but needs to be controllable in a precise manner.

Babel accepts a configuration file to specify this mapping of files to extraction methods, which is described below.

Front-Ends

Babel provides two different front-ends to access its functionality for working with message catalogs:

- A *Command-Line Interface*, and
- *Distutils/Setuptools Integration*

Which one you choose depends on the nature of your project. For most modern Python projects, the distutils/setuptools integration is probably more convenient.

Extraction Method Mapping and Configuration

The mapping of extraction methods to files in Babel is done via a configuration file. This file maps extended glob patterns to the names of the extraction methods, and can also set various options for each pattern (which options are available depends on the specific extraction method).

For example, the following configuration adds extraction of messages from both Genshi markup templates and text templates:

```
# Extraction from Python source files

[python: **.py]

# Extraction from Genshi HTML and text templates

[genshi: **/templates/**.html]
ignore_tags = script,style
include_attrs = alt title summary

[genshi: **/templates/**.txt]
template_class = genshi.template:TextTemplate
encoding = ISO-8819-15

# Extraction from JavaScript files
```

```
[javascript: **.js]
extract_messages = $._, jQuery._
```

The configuration file syntax is based on the format commonly found in `.INI` files on Windows systems, and as supported by the `ConfigParser` module in the Python standard library. Section names (the strings enclosed in square brackets) specify both the name of the extraction method, and the extended glob pattern to specify the files that this extraction method should be used for, separated by a colon. The options in the sections are passed to the extraction method. Which options are available is specific to the extraction method used.

The extended glob patterns used in this configuration are similar to the glob patterns provided by most shells. A single asterisk (`*`) is a wildcard for any number of characters (except for the pathname component separator "/"), while a question mark (`?`) only matches a single character. In addition, two subsequent asterisk characters (`**`) can be used to make the wildcard match any directory level, so the pattern `**.txt` matches any file with the extension `.txt` in any directory.

Lines that start with a `#` or `;` character are ignored and can be used for comments. Empty lines are ignored, too.

Note: if you're performing message extraction using the command Babel provides for integration into `setup.py` scripts, you can also provide this configuration in a different way, namely as a keyword argument to the `setup()` function. See *Distutils/Setuptools Integration* for more information.

Default Extraction Methods

Babel comes with a few builtin extractors: `python` (which extracts messages from Python source files), `javascript`, and `ignore` (which extracts nothing).

The `python` extractor is by default mapped to the glob pattern `**.py`, meaning it'll be applied to all files with the `.py` extension in any directory. If you specify your own mapping configuration, this default mapping is discarded, so you need to explicitly add it to your mapping (as shown in the example above.)

Referencing Extraction Methods

To be able to use short extraction method names such as "genshi", you need to have pkg_resources installed, and the package implementing that extraction method needs to have been installed with its meta data (the egg-info).

If this is not possible for some reason, you need to map the short names to fully qualified function names in an extract section in the mapping configuration. For example:

```
# Some custom extraction method

[extractors]
custom = mypackage.module:extract_custom

[custom: **.ctm]
some_option = foo
```

Note that the builtin extraction methods `python` and `ignore` are available by default, even if pkg_resources is not installed. You should never need to explicitly define them in the `[extractors]` section.

Writing Extraction Methods

Adding new methods for extracting localizable methods is easy. First, you'll need to implement a function that complies with the following interface:

```
def extract_xxx(fileobj, keywords, comment_tags, options):
    """Extract messages from XXX files.

    :param fileobj: the file-like object the messages should be extracted
                    from
    :param keywords: a list of keywords (i.e. function names) that should
                     be recognized as translation functions
    :param comment_tags: a list of translator tags to search for and
                         include in the results
    :param options: a dictionary of additional options (optional)
    :return: an iterator over ``(lineno, funcname, message, comments)``
             tuples
    :rtype: ``iterator``
    """
```

Note: Any strings in the tuples produced by this function must be either `unicode` objects, or `str` objects using plain ASCII characters. That means that if sources contain strings using other encodings, it is the job of the extractor implementation to do the decoding to `unicode` objects.

Next, you should register that function as an entry point. This requires your `setup.py` script to use setuptools, and your package to be installed with the necessary metadata. If that's taken care of, add something like the following to your `setup.py` script:

```
def setup(...

    entry_points = """
    [babel.extractors]
    xxx = your.package:extract_xxx
    """,
```

That is, add your extraction method to the entry point group `babel.extractors`, where the name of the entry point is the name that people will use to reference the extraction method, and the value being the module and the name of the function (separated by a colon) implementing the actual extraction.

Note: As shown in *Referencing Extraction Methods*, declaring an entry point is not strictly required, as users can still reference the extraction function directly. But whenever possible, the entry point should be declared to make configuration more convenient.

Translator Comments

First of all what are comments tags. Comments tags are excerpts of text to search for in comments, only comments, right before the python gettext calls, as shown on the following example:

```
# NOTE: This is a comment about `Foo Bar`
_('Foo Bar')
```

The comments tag for the above example would be `NOTE:`, and the translator comment for that tag would be `This is a comment about ‘Foo Bar‘`.

The resulting output in the catalog template would be something like:

```
#. This is a comment about `Foo Bar`
#: main.py:2
msgid "Foo Bar"
msgstr ""
```

Now, you might ask, why would I need that?

Consider this simple case; you have a menu item called “manual”. You know what it means, but when the translator sees this they will wonder did you mean:

1. a document or help manual, or
2. a manual process?

This is the simplest case where a translation comment such as “The installation manual” helps to clarify the situation and makes a translator more productive.

Note: Whether translator comments can be extracted depends on the extraction method in use. The Python extractor provided by Babel does implement this feature, but others may not.

1.7 Command-Line Interface

Babel includes a command-line interface for working with message catalogs, similar to the various GNU `gettext` tools commonly available on Linux/Unix systems.

When properly installed, Babel provides a script called `pybabel`:

```
$ pybabel --help
usage: pybabel command [options] [args]

options:
  --version       show program's version number and exit
  -h, --help      show this help message and exit
  --list-locales  print all known locales and exit
  -v, --verbose   print as much as possible
  -q, --quiet     print as little as possible

commands:
  compile  compile message catalogs to MO files
  extract  extract messages from source files and generate a POT file
  init     create new message catalogs from a POT file
  update   update existing message catalogs from a POT file
```

The `pybabel` script provides a number of sub-commands that do the actual work. Those sub-commands are described below.

1.7.1 compile

The `compile` sub-command can be used to compile translation catalogs into binary MO files:

```
$ pybabel compile --help
usage: pybabel compile [options]

compile message catalogs to MO files

options:
  -h, --help            show this help message and exit
  -D DOMAIN, --domain=DOMAIN
                        domain of MO and PO files (default 'messages')
  -d DIR, --directory=DIR
                        base directory of catalog files
  -l LOCALE, --locale=LOCALE
                        locale of the catalog
  -i FILE, --input-file=FILE
                        name of the input file
  -o FILE, --output-file=FILE
                        name of the output file (default
                        '<output_dir>/<locale>/LC_MESSAGES/<domain>.mo')
  -f, --use-fuzzy       also include fuzzy translations (default False)
  --statistics          print statistics about translations
```

If `directory` is specified, but `output-file` is not, the default filename of the output file will be:

```
<directory>/<locale>/LC_MESSAGES/<domain>.mo
```

If neither the `input_file` nor the `locale` option is set, this command looks for all catalog files in the base directory that match the given domain, and compiles each of them to MO files in the same directory.

1.7.2 extract

The `extract` sub-command can be used to extract localizable messages from a collection of source files:

```
$ pybabel extract --help
usage: pybabel extract [options] dir1 <dir2> ...

extract messages from source files and generate a POT file

options:
  -h, --help            show this help message and exit
  --charset=CHARSET     charset to use in the output (default "utf-8")
  -k KEYWORDS, --keyword=KEYWORDS
                        keywords to look for in addition to the defaults. You
                        can specify multiple -k flags on the command line.
  --no-default-keywords
                        do not include the default keywords
  -F MAPPING_FILE, --mapping=MAPPING_FILE
                        path to the extraction mapping file
  --no-location         do not include location comments with filename and
                        line number
  --omit-header         do not include msgid "" entry in header
  -o OUTPUT, --output=OUTPUT
                        path to the output POT file
  -w WIDTH, --width=WIDTH
                        set output line width (default 76)
  --no-wrap             do not break long message lines, longer than the
                        output line width, into several lines
  --sort-output         generate sorted output (default False)
```

```
  --sort-by-file        sort output by file location (default False)
  --msgid-bugs-address=EMAIL@ADDRESS
                        set report address for msgid
  --copyright-holder=COPYRIGHT_HOLDER
                        set copyright holder in output
  -c TAG, --add-comments=TAG
                        place comment block with TAG (or those preceding
                        keyword lines) in output file. One TAG per argument
                        call
```

1.7.3 init

The *init* sub-command creates a new translations catalog based on a PO template file:

```
$ pybabel init --help
usage: pybabel init [options]

create new message catalogs from a POT file

options:
  -h, --help            show this help message and exit
  -D DOMAIN, --domain=DOMAIN
                        domain of PO file (default 'messages')
  -i FILE, --input-file=FILE
                        name of the input file
  -d DIR, --output-dir=DIR
                        path to output directory
  -o FILE, --output-file=FILE
                        name of the output file (default
                        '<output_dir>/<locale>/LC_MESSAGES/<domain>.po')
  -l LOCALE, --locale=LOCALE
                        locale for the new localized catalog
```

1.7.4 update

The *update* sub-command updates an existing new translations catalog based on a PO template file:

```
$ pybabel update --help
usage: pybabel update [options]

update existing message catalogs from a POT file

options:
  -h, --help            show this help message and exit
  -D DOMAIN, --domain=DOMAIN
                        domain of PO file (default 'messages')
  -i FILE, --input-file=FILE
                        name of the input file
  -d DIR, --output-dir=DIR
                        path to output directory
  -o FILE, --output-file=FILE
                        name of the output file (default
                        '<output_dir>/<locale>/LC_MESSAGES/<domain>.po')
  -l LOCALE, --locale=LOCALE
                        locale of the translations catalog
```

```
  --ignore-obsolete     do not include obsolete messages in the output
                        (default False)
  -N, --no-fuzzy-matching
                        do not use fuzzy matching (default False)
  --previous            keep previous msgids of translated messages (default
                        False)
```

If `output_dir` is specified, but `output-file` is not, the default filename of the output file will be:

```
<directory>/<locale>/LC_MESSAGES/<domain>.mo
```

If neither the `output_file` nor the `locale` option is set, this command looks for all catalog files in the base directory that match the given domain, and updates each of them.

1.8 Distutils/Setuptools Integration

Babel provides commands for integration into `setup.py` scripts, based on either the `distutils` package that is part of the Python standard library, or the third-party `setuptools` package.

These commands are available by default when Babel has been properly installed, and `setup.py` is using `setuptools`. For projects that use plain old `distutils`, the commands need to be registered explicitly, for example:

```
from distutils.core import setup
from babel.messages import frontend as babel

setup(
    ...
    cmdclass = {'compile_catalog': babel.compile_catalog,
                'extract_messages': babel.extract_messages,
                'init_catalog': babel.init_catalog,
                'update_catalog': babel.update_catalog}
)
```

1.8.1 compile_catalog

The `compile_catalog` command is similar to the GNU `msgfmt` tool, in that it takes a message catalog from a PO file and compiles it to a binary MO file.

If the command has been correctly installed or registered, a project's `setup.py` script should allow you to use the command:

```
$ ./setup.py compile_catalog --help
Global options:
  --verbose (-v)  run verbosely (default)
  --quiet (-q)    run quietly (turns verbosity off)
  --dry-run (-n)  don't actually do anything
  --help (-h)     show detailed help message

Options for 'compile_catalog' command:
   ...
```

Running the command will produce a binary MO file:

```
$ ./setup.py compile_catalog --directory foobar/locale --locale pt_BR
running compile_catalog
compiling catalog to foobar/locale/pt_BR/LC_MESSAGES/messages.mo
```

Options

The `compile_catalog` command accepts the following options:

Option	Description
`--domain`	domain of the PO file (defaults to lower-cased project name)
`--directory` (`-d`)	name of the base directory
`--input-file` (`-i`)	name of the input file
`--output-file` (`-o`)	name of the output file
`--locale` (`-l`)	locale for the new localized string
`--use-fuzzy` (`-f`)	also include "fuzzy" translations
`--statistics`	print statistics about translations

If `directory` is specified, but `output-file` is not, the default filename of the output file will be:

```
<directory>/<locale>/LC_MESSAGES/<domain>.mo
```

If neither the `input_file` nor the `locale` option is set, this command looks for all catalog files in the base directory that match the given domain, and compiles each of them to MO files in the same directory.

These options can either be specified on the command-line, or in the `setup.cfg` file.

1.8.2 extract_messages

The `extract_messages` command is comparable to the GNU `xgettext` program: it can extract localizable messages from a variety of difference source files, and generate a PO (portable object) template file from the collected messages.

If the command has been correctly installed or registered, a project's `setup.py` script should allow you to use the command:

```
$ ./setup.py extract_messages --help
Global options:
  --verbose (-v)  run verbosely (default)
  --quiet (-q)    run quietly (turns verbosity off)
  --dry-run (-n)  don't actually do anything
  --help (-h)     show detailed help message

Options for 'extract_messages' command:
   ...
```

Running the command will produce a PO template file:

```
$ ./setup.py extract_messages --output-file foobar/locale/messages.pot
running extract_messages
extracting messages from foobar/__init__.py
extracting messages from foobar/core.py
...
writing PO template file to foobar/locale/messages.pot
```

Method Mapping

The mapping of file patterns to extraction methods (and options) can be specified using a configuration file that is pointed to using the `--mapping-file` option shown above. Alternatively, you can configure the mapping directly in `setup.py` using a keyword argument to the `setup()` function:

```
setup(...

    message_extractors = {
        'foobar': [
            ('**.py',                'python', None),
            ('**/templates/**.html', 'genshi', None),
            ('**/templates/**.txt',  'genshi', {
                'template_class': 'genshi.template:TextTemplate'
            })
        ],
    },

    ...
)
```

Options

The `extract_messages` command accepts the following options:

Option	Description
`--charset`	charset to use in the output file
`--keywords (-k)`	space-separated list of keywords to look for in addition to the defaults
`--no-default-keywords`	do not include the default keywords
`--mapping-file (-F)`	path to the mapping configuration file
`--no-location`	do not include location comments with filename and line number
`--omit-header`	do not include msgid "" entry in header
`--output-file (-o)`	name of the output file
`--width (-w)`	set output line width (default 76)
`--no-wrap`	do not break long message lines, longer than the output line width, into several lines
`--input-dirs`	directories that should be scanned for messages
`--sort-output`	generate sorted output (default False)
`--sort-by-file`	sort output by file location (default False)
`--msgid-bugs-address`	set email address for message bug reports
`--copyright-holder`	set copyright holder in output
`--add-comments (-c)`	place comment block with TAG (or those preceding keyword lines) in output file. Separate multiple TAGs with commas(,)

These options can either be specified on the command-line, or in the `setup.cfg` file. In the latter case, the options above become entries of the section `[extract_messages]`, and the option names are changed to use underscore characters instead of dashes, for example:

```
[extract_messages]
keywords = _ gettext ngettext
mapping_file = mapping.cfg
width = 80
```

This would be equivalent to invoking the command from the command-line as follows:

```
$ setup.py extract_messages -k _ -k gettext -k ngettext -F mapping.cfg -w 80
```

Any path names are interpreted relative to the location of the `setup.py` file. For boolean options, use "true" or "false" values.

1.8.3 init_catalog

The `init_catalog` command is basically equivalent to the GNU `msginit` program: it creates a new translation catalog based on a PO template file (POT).

If the command has been correctly installed or registered, a project's `setup.py` script should allow you to use the command:

```
$ ./setup.py init_catalog --help
Global options:
  --verbose (-v)  run verbosely (default)
  --quiet (-q)    run quietly (turns verbosity off)
  --dry-run (-n)  don't actually do anything
  --help (-h)     show detailed help message

Options for 'init_catalog' command:
  ...
```

Running the command will produce a PO file:

```
$ ./setup.py init_catalog -l fr -i foobar/locales/messages.pot \
                          -o foobar/locales/fr/messages.po
running init_catalog
creating catalog 'foobar/locales/fr/messages.po' based on 'foobar/locales/messages.pot'
```

Options

The `init_catalog` command accepts the following options:

Option	Description
`--domain`	domain of the PO file (defaults to lower-cased project name)
`--input-file` (`-i`)	name of the input file
`--output-dir` (`-d`)	name of the output directory
`--output-file` (`-o`)	name of the output file
`--locale`	locale for the new localized string

If `output-dir` is specified, but `output-file` is not, the default filename of the output file will be:

```
<output_dir>/<locale>/LC_MESSAGES/<domain>.po
```

These options can either be specified on the command-line, or in the `setup.cfg` file.

1.8.4 update_catalog

The `update_catalog` command is basically equivalent to the GNU `msgmerge` program: it updates an existing translations catalog based on a PO template file (POT).

If the command has been correctly installed or registered, a project's `setup.py` script should allow you to use the command:

```
$ ./setup.py update_catalog --help
Global options:
  --verbose (-v)  run verbosely (default)
  --quiet (-q)    run quietly (turns verbosity off)
  --dry-run (-n)  don't actually do anything
  --help (-h)     show detailed help message

Options for 'update_catalog' command:
  ...
```

Running the command will update a PO file:

```
$ ./setup.py update_catalog -l fr -i foobar/locales/messages.pot \
                            -o foobar/locales/fr/messages.po
running update_catalog
updating catalog 'foobar/locales/fr/messages.po' based on 'foobar/locales/messages.pot'
```

Options

The `update_catalog` command accepts the following options:

Option	Description
`--domain`	domain of the PO file (defaults to lower-cased project name)
`--input-file` (`-i`)	name of the input file
`--output-dir` (`-d`)	name of the output directory
`--output-file` (`-o`)	name of the output file
`--locale`	locale for the new localized string
`--ignore-obsolete`	do not include obsolete messages in the output
`--no-fuzzy-matching` (`-N`)	do not use fuzzy matching
`--previous`	keep previous msgids of translated messages

If `output-dir` is specified, but `output-file` is not, the default filename of the output file will be:

```
<output_dir>/<locale>/LC_MESSAGES/<domain>.po
```

If neither the `input_file` nor the `locale` option is set, this command looks for all catalog files in the base directory that match the given domain, and updates each of them.

These options can either be specified on the command-line, or in the `setup.cfg` file.

1.9 Support Classes and Functions

The `babel.support` modules contains a number of classes and functions that can help with integrating Babel, and internationalization in general, into your application or framework. The code in this module is not used by Babel itself, but instead is provided to address common requirements of applications that should handle internationalization.

1.9.1 Lazy Evaluation

One such requirement is lazy evaluation of translations. Many web-based applications define some localizable message at the module level, or in general at some level where the locale of the remote user is not yet known. For such cases, web frameworks generally provide a "lazy" variant of the `gettext` functions, which basically translates the message not when the `gettext` function is invoked, but when the string is accessed in some manner.

1.9.2 Extended Translations Class

Many web-based applications are composed of a variety of different components (possibly using some kind of plugin system), and some of those components may provide their own message catalogs that need to be integrated into the larger system.

To support this usage pattern, Babel provides a `Translations` class that is derived from the `GNUTranslations` class in the `gettext` module. This class adds a `merge()` method that takes another `Translations` instance, and merges the content of the latter into the main catalog:

```
translations = Translations.load('main')
translations.merge(Translations.load('plugin1'))
```

CHAPTER 2

API Reference

The API reference lists the full public API that Babel provides.

2.1 API Reference

This part of the documentation contains the full API reference of the public API of Babel.

2.1.1 Core Functionality

The core API provides the basic core functionality. Primarily it provides the *Locale* object and ways to create it. This object encapsulates a locale and exposes all the data it contains.

All the core functionality is also directly importable from the *babel* module for convenience.

Basic Interface

class babel.core.Locale(*language*, *territory=None*, *script=None*, *variant=None*)

Representation of a specific locale.

```
>>> locale = Locale('en', 'US')
>>> repr(locale)
"Locale('en', territory='US')"
>>> locale.display_name
u'English (United States)'
```

A *Locale* object can also be instantiated from a raw locale string:

```
>>> locale = Locale.parse('en-US', sep='-')
>>> repr(locale)
"Locale('en', territory='US')"
```

Locale objects provide access to a collection of locale data, such as territory and language names, number and date format patterns, and more:

```
>>> locale.number_symbols['decimal']
u'.'
```

If a locale is requested for which no locale data is available, an *UnknownLocaleError* is raised:

```
>>> Locale.parse('en_XX')
Traceback (most recent call last):
    ...
UnknownLocaleError: unknown locale 'en_XX'
```

For more information see **RFC 3066**.

`currencies`
: Mapping of currency codes to translated currency names. This only returns the generic form of the currency name, not the count specific one. If an actual number is requested use the *babel.numbers.get_currency_name()* function.

```
>>> Locale('en').currencies['COP']
u'Colombian Peso'
>>> Locale('de', 'DE').currencies['COP']
u'Kolumbianischer Peso'
```

`currency_formats`
: Locale patterns for currency number formatting.

Note: The format of the value returned may change between Babel versions.

```
>>> Locale('en', 'US').currency_formats['standard']
<NumberPattern u'\xa4#,##0.00'>
>>> Locale('en', 'US').currency_formats['accounting']
<NumberPattern u'\xa4#,##0.00'>
```

`currency_symbols`
: Mapping of currency codes to symbols.

```
>>> Locale('en', 'US').currency_symbols['USD']
u'$'
>>> Locale('es', 'CO').currency_symbols['USD']
u'US$'
```

`date_formats`
: Locale patterns for date formatting.

Note: The format of the value returned may change between Babel versions.

```
>>> Locale('en', 'US').date_formats['short']
<DateTimePattern u'M/d/yy'>
>>> Locale('fr', 'FR').date_formats['long']
<DateTimePattern u'd MMMM y'>
```

`datetime_formats`
: Locale patterns for datetime formatting.

Note: The format of the value returned may change between Babel versions.

```
>>> Locale('en').datetime_formats['full']
u"{1} 'at' {0}"
>>> Locale('th').datetime_formats['medium']
u'{1} {0}'
```

`datetime_skeletons`
Locale patterns for formatting parts of a datetime.

```
>>> Locale('en').datetime_skeletons['MEd']
<DateTimePattern u'E, M/d'>
>>> Locale('fr').datetime_skeletons['MEd']
<DateTimePattern u'E dd/MM'>
>>> Locale('fr').datetime_skeletons['H']
<DateTimePattern u"HH 'h'">
```

`days`
Locale display names for weekdays.

```
>>> Locale('de', 'DE').days['format']['wide'][3]
u'Donnerstag'
```

`decimal_formats`
Locale patterns for decimal number formatting.

Note: The format of the value returned may change between Babel versions.

```
>>> Locale('en', 'US').decimal_formats[None]
<NumberPattern u'#,##0.###'>
```

classmethod `default`(*category=None*, *aliases={'el': 'el_GR', 'fi': 'fi_FI', 'bg': 'bg_BG', 'uk': 'uk_UA', 'tr': 'tr_TR', 'ca': 'ca_ES', 'de': 'de_DE', 'fr': 'fr_FR', 'da': 'da_DK', 'fa': 'fa_IR', 'ar': 'ar_SY', 'mk': 'mk_MK', 'bs': 'bs_BA', 'cs': 'cs_CZ', 'et': 'et_EE', 'gl': 'gl_ES', 'id': 'id_ID', 'es': 'es_ES', 'he': 'he_IL', 'ru': 'ru_RU', 'nl': 'nl_NL', 'pt': 'pt_PT', 'nn': 'nn_NO', 'no': 'nb_NO', 'ko': 'ko_KR', 'sv': 'sv_SE', 'km': 'km_KH', 'ja': 'ja_JP', 'lv': 'lv_LV', 'lt': 'lt_LT', 'en': 'en_US', 'sk': 'sk_SK', 'th': 'th_TH', 'sl': 'sl_SI', 'it': 'it_IT', 'hu': 'hu_HU', 'ro': 'ro_RO', 'is': 'is_IS', 'pl': 'pl_PL'}*)
Return the system default locale for the specified category.

```
>>> for name in ['LANGUAGE', 'LC_ALL', 'LC_CTYPE', 'LC_MESSAGES']:
...     os.environ[name] = ''
>>> os.environ['LANG'] = 'fr_FR.UTF-8'
>>> Locale.default('LC_MESSAGES')
Locale('fr', territory='FR')
```

The following fallbacks to the variable are always considered:

- `LANGUAGE`
- `LC_ALL`
- `LC_CTYPE`
- `LANG`

Parameters

- `category` – one of the `LC_XXX` environment variable names
- `aliases` – a dictionary of aliases for locale identifiers

`display_name`
The localized display name of the locale.

```
>>> Locale('en').display_name
u'English'
>>> Locale('en', 'US').display_name
u'English (United States)'
>>> Locale('sv').display_name
u'svenska'
```

Type *unicode*

`english_name`
The english display name of the locale.

```
>>> Locale('de').english_name
u'German'
>>> Locale('de', 'DE').english_name
u'German (Germany)'
```

Type *unicode*

`eras`
Locale display names for eras.

Note: The format of the value returned may change between Babel versions.

```
>>> Locale('en', 'US').eras['wide'][1]
u'Anno Domini'
>>> Locale('en', 'US').eras['abbreviated'][0]
u'BC'
```

`first_week_day`
The first day of a week, with 0 being Monday.

```
>>> Locale('de', 'DE').first_week_day
0
>>> Locale('en', 'US').first_week_day
6
```

`get_display_name(`*locale=None*`)`
Return the display name of the locale using the given locale.

The display name will include the language, territory, script, and variant, if those are specified.

```
>>> Locale('zh', 'CN', script='Hans').get_display_name('en')
u'Chinese (Simplified, China)'
```

Parameters `locale` – the locale to use

`get_language_name(`*locale=None*`)`
Return the language of this locale in the given locale.

```
>>> Locale('zh', 'CN', script='Hans').get_language_name('de')
u'Chinesisch'
```

New in version 1.0.

Parameters `locale` – the locale to use

`get_script_name`(*locale=None*)
Return the script name in the given locale.

`get_territory_name`(*locale=None*)
Return the territory name in the given locale.

`interval_formats`
Locale patterns for interval formatting.

Note: The format of the value returned may change between Babel versions.

How to format date intervals in Finnish when the day is the smallest changing component:

```
>>> Locale('fi_FI').interval_formats['MEd']['d']
[u'E d. \u2013 ', u'E d.M.']
```

See also:

The primary API to use this data is *babel.dates.format_interval()*.

Return type dict[str, dict[str, list[str]]]

`language` = **None**
the language code

`language_name`
The localized language name of the locale.

```
>>> Locale('en', 'US').language_name
u'English'
```

`languages`
Mapping of language codes to translated language names.

```
>>> Locale('de', 'DE').languages['ja']
u'Japanisch'
```

See ISO 639 for more information.

`list_patterns`
Patterns for generating lists

Note: The format of the value returned may change between Babel versions.

```
>>> Locale('en').list_patterns['start']
u'{0}, {1}'
>>> Locale('en').list_patterns['end']
u'{0}, and {1}'
>>> Locale('en_GB').list_patterns['end']
u'{0} and {1}'
```

`meta_zones`
Locale display names for meta time zones.

Meta time zones are basically groups of different Olson time zones that have the same GMT offset and daylight savings time.

Note: The format of the value returned may change between Babel versions.

```
>>> Locale('en', 'US').meta_zones['Europe_Central']['long']['daylight']
u'Central European Summer Time'
```

New in version 0.9.

`min_week_days`
The minimum number of days in a week so that the week is counted as the first week of a year or month.

```
>>> Locale('de', 'DE').min_week_days
4
```

`months`
Locale display names for months.

```
>>> Locale('de', 'DE').months['format']['wide'][10]
u'Oktober'
```

classmethod `negotiate`(*preferred*, *available*, *sep='_'*, *aliases={'el': 'el_GR', 'fi': 'fi_FI', 'bg': 'bg_BG', 'uk': 'uk_UA', 'tr': 'tr_TR', 'ca': 'ca_ES', 'de': 'de_DE', 'fr': 'fr_FR', 'da': 'da_DK', 'fa': 'fa_IR', 'ar': 'ar_SY', 'mk': 'mk_MK', 'bs': 'bs_BA', 'cs': 'cs_CZ', 'et': 'et_EE', 'gl': 'gl_ES', 'id': 'id_ID', 'es': 'es_ES', 'he': 'he_IL', 'ru': 'ru_RU', 'nl': 'nl_NL', 'pt': 'pt_PT', 'nn': 'nn_NO', 'no': 'nb_NO', 'ko': 'ko_KR', 'sv': 'sv_SE', 'km': 'km_KH', 'ja': 'ja_JP', 'lv': 'lv_LV', 'lt': 'lt_LT', 'en': 'en_US', 'sk': 'sk_SK', 'th': 'th_TH', 'sl': 'sl_SI', 'it': 'it_IT', 'hu': 'hu_HU', 'ro': 'ro_RO', 'is': 'is_IS', 'pl': 'pl_PL'}*)
Find the best match between available and requested locale strings.

```
>>> Locale.negotiate(['de_DE', 'en_US'], ['de_DE', 'de_AT'])
Locale('de', territory='DE')
>>> Locale.negotiate(['de_DE', 'en_US'], ['en', 'de'])
Locale('de')
>>> Locale.negotiate(['de_DE', 'de'], ['en_US'])
```

You can specify the character used in the locale identifiers to separate the differnet components. This separator is applied to both lists. Also, case is ignored in the comparison:

```
>>> Locale.negotiate(['de-DE', 'de'], ['en-us', 'de-de'], sep='-')
Locale('de', territory='DE')
```

Parameters

- `preferred` – the list of locale identifers preferred by the user
- `available` – the list of locale identifiers available
- `aliases` – a dictionary of aliases for locale identifiers

`number_symbols`
Symbols used in number formatting.

Note: The format of the value returned may change between Babel versions.

```
>>> Locale('fr', 'FR').number_symbols['decimal']
u','
```

`ordinal_form`
Plural rules for the locale.

```
>>> Locale('en').ordinal_form(1)
'one'
>>> Locale('en').ordinal_form(2)
'two'
>>> Locale('en').ordinal_form(3)
'few'
>>> Locale('fr').ordinal_form(2)
'other'
>>> Locale('ru').ordinal_form(100)
'other'
```

classmethod `parse`(*identifier*, *sep='_'*, *resolve_likely_subtags=True*)
Create a *Locale* instance for the given locale identifier.

```
>>> l = Locale.parse('de-DE', sep='-')
>>> l.display_name
u'Deutsch (Deutschland)'
```

If the *identifier* parameter is not a string, but actually a *Locale* object, that object is returned:

```
>>> Locale.parse(l)
Locale('de', territory='DE')
```

This also can perform resolving of likely subtags which it does by default. This is for instance useful to figure out the most likely locale for a territory you can use `'und'` as the language tag:

```
>>> Locale.parse('und_AT')
Locale('de', territory='AT')
```

Parameters

- `identifier` – the locale identifier string
- `sep` – optional component separator
- `resolve_likely_subtags` – if this is specified then a locale will have its likely subtag resolved if the locale otherwise does not exist. For instance `zh_TW` by itself is not a locale that exists but Babel can automatically expand it to the full form of `zh_hant_TW`. Note that this expansion is only taking place if no locale exists otherwise. For instance there is a locale `en` that can exist by itself.

Raises

- `ValueError` – if the string does not appear to be a valid locale identifier
- `UnknownLocaleError` – if no locale data is available for the requested locale

`percent_formats`
Locale patterns for percent number formatting.

Note: The format of the value returned may change between Babel versions.

```
>>> Locale('en', 'US').percent_formats[None]
<NumberPattern u'#,##0%'>
```

`periods`
Locale display names for day periods (AM/PM).

```
>>> Locale('en', 'US').periods['am']
u'AM'
```

`plural_form`
Plural rules for the locale.

```
>>> Locale('en').plural_form(1)
'one'
>>> Locale('en').plural_form(0)
'other'
>>> Locale('fr').plural_form(0)
'one'
>>> Locale('ru').plural_form(100)
'many'
```

`quarters`
Locale display names for quarters.

```
>>> Locale('de', 'DE').quarters['format']['wide'][1]
u'1. Quartal'
```

`scientific_formats`
Locale patterns for scientific number formatting.

Note: The format of the value returned may change between Babel versions.

```
>>> Locale('en', 'US').scientific_formats[None]
<NumberPattern u'#E0'>
```

`script` = **None**
the script code

`script_name`
The localized script name of the locale if available.

```
>>> Locale('sr', 'ME', script='Latn').script_name
u'latinica'
```

`scripts`
Mapping of script codes to translated script names.

```
>>> Locale('en', 'US').scripts['Hira']
u'Hiragana'
```

See ISO 15924 for more information.

`territories`
Mapping of script codes to translated script names.

```
>>> Locale('es', 'CO').territories['DE']
u'Alemania'
```

See ISO 3166 for more information.

territory = None
: the territory (country or region) code

territory_name
: The localized territory name of the locale if available.

```
>>> Locale('de', 'DE').territory_name
u'Deutschland'
```

time_formats
: Locale patterns for time formatting.

Note: The format of the value returned may change between Babel versions.

```
>>> Locale('en', 'US').time_formats['short']
<DateTimePattern u'h:mm a'>
>>> Locale('fr', 'FR').time_formats['long']
<DateTimePattern u'HH:mm:ss z'>
```

time_zones
: Locale display names for time zones.

Note: The format of the value returned may change between Babel versions.

```
>>> Locale('en', 'US').time_zones['Europe/London']['long']['daylight']
u'British Summer Time'
>>> Locale('en', 'US').time_zones['America/St_Johns']['city']
u'St. John\u2019s'
```

variant = None
: the variant code

variants
: Mapping of script codes to translated script names.

```
>>> Locale('de', 'DE').variants['1901']
u'Alte deutsche Rechtschreibung'
```

weekend_end
: The day the weekend ends, with 0 being Monday.

```
>>> Locale('de', 'DE').weekend_end
6
```

weekend_start
: The day the weekend starts, with 0 being Monday.

```
>>> Locale('de', 'DE').weekend_start
5
```

zone_formats
: Patterns related to the formatting of time zones.

Note: The format of the value returned may change between Babel versions.

```
>>> Locale('en', 'US').zone_formats['fallback']
u'%(1)s (%(0)s)'
>>> Locale('pt', 'BR').zone_formats['region']
u'Hor\xe1rio %s'
```

New in version 0.9.

babel.core.default_locale(*category=None*, *aliases={'el': 'el_GR', 'fi': 'fi_FI', 'bg': 'bg_BG', 'uk': 'uk_UA', 'tr': 'tr_TR', 'ca': 'ca_ES', 'de': 'de_DE', 'fr': 'fr_FR', 'da': 'da_DK', 'fa': 'fa_IR', 'ar': 'ar_SY', 'mk': 'mk_MK', 'bs': 'bs_BA', 'cs': 'cs_CZ', 'et': 'et_EE', 'gl': 'gl_ES', 'id': 'id_ID', 'es': 'es_ES', 'he': 'he_IL', 'ru': 'ru_RU', 'nl': 'nl_NL', 'pt': 'pt_PT', 'nn': 'nn_NO', 'no': 'nb_NO', 'ko': 'ko_KR', 'sv': 'sv_SE', 'km': 'km_KH', 'ja': 'ja_JP', 'lv': 'lv_LV', 'lt': 'lt_LT', 'en': 'en_US', 'sk': 'sk_SK', 'th': 'th_TH', 'sl': 'sl_SI', 'it': 'it_IT', 'hu': 'hu_HU', 'ro': 'ro_RO', 'is': 'is_IS', 'pl': 'pl_PL'}*)

Returns the system default locale for a given category, based on environment variables.

```
>>> for name in ['LANGUAGE', 'LC_ALL', 'LC_CTYPE']:
...     os.environ[name] = ''
>>> os.environ['LANG'] = 'fr_FR.UTF-8'
>>> default_locale('LC_MESSAGES')
'fr_FR'
```

The "C" or "POSIX" pseudo-locales are treated as aliases for the "en_US_POSIX" locale:

```
>>> os.environ['LC_MESSAGES'] = 'POSIX'
>>> default_locale('LC_MESSAGES')
'en_US_POSIX'
```

The following fallbacks to the variable are always considered:

- LANGUAGE
- LC_ALL
- LC_CTYPE
- LANG

Parameters

- `category` – one of the `LC_XXX` environment variable names
- `aliases` – a dictionary of aliases for locale identifiers

babel.core.negotiate_locale(*preferred*, *available*, *sep='_'*, *aliases={'el': 'el_GR', 'fi': 'fi_FI', 'bg': 'bg_BG', 'uk': 'uk_UA', 'tr': 'tr_TR', 'ca': 'ca_ES', 'de': 'de_DE', 'fr': 'fr_FR', 'da': 'da_DK', 'fa': 'fa_IR', 'ar': 'ar_SY', 'mk': 'mk_MK', 'bs': 'bs_BA', 'cs': 'cs_CZ', 'et': 'et_EE', 'gl': 'gl_ES', 'id': 'id_ID', 'es': 'es_ES', 'he': 'he_IL', 'ru': 'ru_RU', 'nl': 'nl_NL', 'pt': 'pt_PT', 'nn': 'nn_NO', 'no': 'nb_NO', 'ko': 'ko_KR', 'sv': 'sv_SE', 'km': 'km_KH', 'ja': 'ja_JP', 'lv': 'lv_LV', 'lt': 'lt_LT', 'en': 'en_US', 'sk': 'sk_SK', 'th': 'th_TH', 'sl': 'sl_SI', 'it': 'it_IT', 'hu': 'hu_HU', 'ro': 'ro_RO', 'is': 'is_IS', 'pl': 'pl_PL'}*)

Find the best match between available and requested locale strings.

```
>>> negotiate_locale(['de_DE', 'en_US'], ['de_DE', 'de_AT'])
'de_DE'
>>> negotiate_locale(['de_DE', 'en_US'], ['en', 'de'])
'de'
```

Case is ignored by the algorithm, the result uses the case of the preferred locale identifier:

```
>>> negotiate_locale(['de_DE', 'en_US'], ['de_de', 'de_at'])
'de_DE'
```

```
>>> negotiate_locale(['de_DE', 'en_US'], ['de_de', 'de_at'])
'de_DE'
```

By default, some web browsers unfortunately do not include the territory in the locale identifier for many locales, and some don't even allow the user to easily add the territory. So while you may prefer using qualified locale identifiers in your web-application, they would not normally match the language-only locale sent by such browsers. To workaround that, this function uses a default mapping of commonly used langauge-only locale identifiers to identifiers including the territory:

```
>>> negotiate_locale(['ja', 'en_US'], ['ja_JP', 'en_US'])
'ja_JP'
```

Some browsers even use an incorrect or outdated language code, such as "no" for Norwegian, where the correct locale identifier would actually be "nb_NO" (Bokmål) or "nn_NO" (Nynorsk). The aliases are intended to take care of such cases, too:

```
>>> negotiate_locale(['no', 'sv'], ['nb_NO', 'sv_SE'])
'nb_NO'
```

You can override this default mapping by passing a different *aliases* dictionary to this function, or you can bypass the behavior althogher by setting the *aliases* parameter to *None*.

Parameters

- `preferred` – the list of locale strings preferred by the user
- `available` – the list of locale strings available
- `sep` – character that separates the different parts of the locale strings
- `aliases` – a dictionary of aliases for locale identifiers

Exceptions

exception `babel.core.UnknownLocaleError`(*identifier*)

Exception thrown when a locale is requested for which no locale data is available.

`identifier` = **None**

The identifier of the locale that could not be found.

Utility Functions

`babel.core.get_global`(*key*)

Return the dictionary for the given key in the global data.

The global data is stored in the `babel/global.dat` file and contains information independent of individual locales.

```
>>> get_global('zone_aliases')['UTC']
u'Etc/GMT'
>>> get_global('zone_territories')['Europe/Berlin']
u'DE'
```

The keys available are:

- currency_fractions
- language_aliases
- likely_subtags
- parent_exceptions
- script_aliases
- territory_aliases
- territory_currencies
- territory_languages
- territory_zones
- variant_aliases
- win_mapping
- zone_aliases
- zone_territories

Note: The internal structure of the data may change between versions.

New in version 0.9.

Parameters `key` – the data key

`babel.core.parse_locale(`*identifier*, *sep='_'*`)`

Parse a locale identifier into a tuple of the form `(language, territory, script, variant)`.

```
>>> parse_locale('zh_CN')
('zh', 'CN', None, None)
>>> parse_locale('zh_Hans_CN')
('zh', 'CN', 'Hans', None)
```

The default component separator is "_", but a different separator can be specified using the *sep* parameter:

```
>>> parse_locale('zh-CN', sep='-')
('zh', 'CN', None, None)
```

If the identifier cannot be parsed into a locale, a *ValueError* exception is raised:

```
>>> parse_locale('not_a_LOCALE_String')
Traceback (most recent call last):
  ...
ValueError: 'not_a_LOCALE_String' is not a valid locale identifier
```

Encoding information and locale modifiers are removed from the identifier:

```
>>> parse_locale('it_IT@euro')
('it', 'IT', None, None)
>>> parse_locale('en_US.UTF-8')
('en', 'US', None, None)
>>> parse_locale('de_DE.iso885915@euro')
('de', 'DE', None, None)
```

See **RFC 4646** for more information.

Parameters

- `identifier` – the locale identifier string
- `sep` – character that separates the different components of the locale identifier

Raises `ValueError` – if the string does not appear to be a valid locale identifier

`babel.core.get_locale_identifier(`*tup*, *sep='_'*`)`

The reverse of *parse_locale()*. It creates a locale identifier out of a `(language, territory, script, variant)` tuple. Items can be set to `None` and trailing `None`s can also be left out of the tuple.

```
>>> get_locale_identifier(('de', 'DE', None, '1999'))
'de_DE_1999'
```

New in version 1.0.

Parameters

- `tup` – the tuple as returned by *parse_locale()*.
- `sep` – the separator for the identifier.

2.1.2 Date and Time

The date and time functionality provided by Babel lets you format standard Python *datetime*, *date* and *time* objects and and work with timezones.

Date and Time Formatting

`babel.dates.format_datetime(`*datetime=None*, *format='medium'*, *tzinfo=None*, *locale=None*`)`

Return a date formatted according to the given pattern.

```
>>> dt = datetime(2007, 4, 1, 15, 30)
>>> format_datetime(dt, locale='en_US')
u'Apr 1, 2007, 3:30:00 PM'
```

For any pattern requiring the display of the time-zone, the third-party `pytz` package is needed to explicitly specify the time-zone:

```
>>> format_datetime(dt, 'full', tzinfo=get_timezone('Europe/Paris'),
...                 locale='fr_FR')
u'dimanche 1 avril 2007 \xe0 17:30:00 heure d\u2019\xe9t\xe9 d\u2019Europe centrale'
>>> format_datetime(dt, "yyyy.MM.dd G 'at' HH:mm:ss zzz",
...                 tzinfo=get_timezone('US/Eastern'), locale='en')
u'2007.04.01 AD at 11:30:00 EDT'
```

Parameters

- `datetime` – the *datetime* object; if *None*, the current date and time is used
- `format` – one of "full", "long", "medium", or "short", or a custom date/time pattern
- `tzinfo` – the timezone to apply to the time for display
- `locale` – a *Locale* object or a locale identifier

`babel.dates.format_date`(*date=None*, *format='medium'*, *locale=None*)

Return a date formatted according to the given pattern.

```
>>> d = date(2007, 4, 1)
>>> format_date(d, locale='en_US')
u'Apr 1, 2007'
>>> format_date(d, format='full', locale='de_DE')
u'Sonntag, 1. April 2007'
```

If you don't want to use the locale default formats, you can specify a custom date pattern:

```
>>> format_date(d, "EEE, MMM d, ''yy", locale='en')
u"Sun, Apr 1, '07"
```

Parameters

- `date` – the `date` or `datetime` object; if *None*, the current date is used
- `format` – one of "full", "long", "medium", or "short", or a custom date/time pattern
- `locale` – a *Locale* object or a locale identifier

`babel.dates.format_time`(*time=None*, *format='medium'*, *tzinfo=None*, *locale=None*)

Return a time formatted according to the given pattern.

```
>>> t = time(15, 30)
>>> format_time(t, locale='en_US')
u'3:30:00 PM'
>>> format_time(t, format='short', locale='de_DE')
u'15:30'
```

If you don't want to use the locale default formats, you can specify a custom time pattern:

```
>>> format_time(t, "hh 'o''clock' a", locale='en')
u"03 o'clock PM"
```

For any pattern requiring the display of the time-zone a timezone has to be specified explicitly:

```
>>> t = datetime(2007, 4, 1, 15, 30)
>>> tzinfo = get_timezone('Europe/Paris')
>>> t = tzinfo.localize(t)
>>> format_time(t, format='full', tzinfo=tzinfo, locale='fr_FR')
u'15:30:00 heure d\u2019\xe9t\xe9 d\u2019Europe centrale'
>>> format_time(t, "hh 'o''clock' a, zzzz", tzinfo=get_timezone('US/Eastern'),
...             locale='en')
u"09 o'clock AM, Eastern Daylight Time"
```

As that example shows, when this function gets passed a `datetime.datetime` value, the actual time in the formatted string is adjusted to the timezone specified by the *tzinfo* parameter. If the `datetime` is "naive" (i.e. it has no associated timezone information), it is assumed to be in UTC.

These timezone calculations are **not** performed if the value is of type `datetime.time`, as without date information there's no way to determine what a given time would translate to in a different timezone

without information about whether daylight savings time is in effect or not. This means that time values are left as-is, and the value of the *tzinfo* parameter is only used to display the timezone name if needed:

```
>>> t = time(15, 30)
>>> format_time(t, format='full', tzinfo=get_timezone('Europe/Paris'),
...             locale='fr_FR')
u'15:30:00 heure normale d\u2019Europe centrale'
>>> format_time(t, format='full', tzinfo=get_timezone('US/Eastern'),
...             locale='en_US')
u'3:30:00 PM Eastern Standard Time'
```

Parameters

- `time` – the `time` or `datetime` object; if *None*, the current time in UTC is used
- `format` – one of "full", "long", "medium", or "short", or a custom date/time pattern
- `tzinfo` – the time-zone to apply to the time for display
- `locale` – a *Locale* object or a locale identifier

`babel.dates.format_timedelta`(*delta*, *granularity='second'*, *threshold=0.85*, *add_direction=False*, *format='long'*, *locale=None*)

Return a time delta according to the rules of the given locale.

```
>>> format_timedelta(timedelta(weeks=12), locale='en_US')
u'3 months'
>>> format_timedelta(timedelta(seconds=1), locale='es')
u'1 segundo'
```

The granularity parameter can be provided to alter the lowest unit presented, which defaults to a second.

```
>>> format_timedelta(timedelta(hours=3), granularity='day',
...                  locale='en_US')
u'1 day'
```

The threshold parameter can be used to determine at which value the presentation switches to the next higher unit. A higher threshold factor means the presentation will switch later. For example:

```
>>> format_timedelta(timedelta(hours=23), threshold=0.9, locale='en_US')
u'1 day'
>>> format_timedelta(timedelta(hours=23), threshold=1.1, locale='en_US')
u'23 hours'
```

In addition directional information can be provided that informs the user if the date is in the past or in the future:

```
>>> format_timedelta(timedelta(hours=1), add_direction=True, locale='en')
u'in 1 hour'
>>> format_timedelta(timedelta(hours=-1), add_direction=True, locale='en')
u'1 hour ago'
```

The format parameter controls how compact or wide the presentation is:

```
>>> format_timedelta(timedelta(hours=3), format='short', locale='en')
u'3 hr'
>>> format_timedelta(timedelta(hours=3), format='narrow', locale='en')
u'3h'
```

Parameters

- `delta` – a `timedelta` object representing the time difference to format, or the delta in seconds as an *int* value
- `granularity` – determines the smallest unit that should be displayed, the value can be one of "year", "month", "week", "day", "hour", "minute" or "second"
- `threshold` – factor that determines at which point the presentation switches to the next higher unit
- `add_direction` – if this flag is set to *True* the return value will include directional information. For instance a positive timedelta will include the information about it being in the future, a negative will be information about the value being in the past.
- `format` – the format, can be "narrow", "short" or "long". ("medium" is deprecated, currently converted to "long" to maintain compatibility)
- `locale` – a *Locale* object or a locale identifier

`babel.dates.format_skeleton(`*skeleton*, *datetime=None*, *tzinfo=None*, *fuzzy=True*, *locale=None*`)`
Return a time and/or date formatted according to the given pattern.

The skeletons are defined in the CLDR data and provide more flexibility than the simple short/long/medium formats, but are a bit harder to use. The are defined using the date/time symbols without order or punctuation and map to a suitable format for the given locale.

```
>>> t = datetime(2007, 4, 1, 15, 30)
>>> format_skeleton('MMMEd', t, locale='fr')
u'dim. 1 avr.'
>>> format_skeleton('MMMEd', t, locale='en')
u'Sun, Apr 1'
>>> format_skeleton('yMMd', t, locale='fi')  # yMMd is not in the Finnish locale; yMd gets used
u'1.4.2007'
>>> format_skeleton('yMMd', t, fuzzy=False, locale='fi')  # yMMd is not in the Finnish locale, an error is th
Traceback (most recent call last):
    ...
KeyError: yMMd
```

After the skeleton is resolved to a pattern *format_ datetime* is called so all timezone processing etc is the same as for that.

Parameters

- `skeleton` – A date time skeleton as defined in the cldr data.
- `datetime` – the `time` or `datetime` object; if *None*, the current time in UTC is used
- `tzinfo` – the time-zone to apply to the time for display
- `fuzzy` – If the skeleton is not found, allow choosing a skeleton that's close enough to it.
- `locale` – a *Locale* object or a locale identifier

`babel.dates.format_interval(`*start*, *end*, *skeleton=None*, *tzinfo=None*, *fuzzy=True*, *locale=None*`)`
Format an interval between two instants according to the locale's rules.

```
>>> format_interval(date(2016, 1, 15), date(2016, 1, 17), "yMd", locale="fi")
u'15.\u201317.1.2016'
```

```
>>> format_interval(time(12, 12), time(16, 16), "Hm", locale="en_GB")
'12:12 \u2013 16:16'
```

```
>>> format_interval(time(5, 12), time(16, 16), "hm", locale="en_US")
'5:12 AM \u2013 4:16 PM'
```

```
>>> format_interval(time(16, 18), time(16, 24), "Hm", locale="it")
'16:18\u201316:24'
```

If the start instant equals the end instant, the interval is formatted like the instant.

```
>>> format_interval(time(16, 18), time(16, 18), "Hm", locale="it")
'16:18'
```

Unknown skeletons fall back to "default" formatting.

```
>>> format_interval(date(2015, 1, 1), date(2017, 1, 1), "wzq", locale="ja")
'2015/01/01\uff5e2017/01/01'
```

```
>>> format_interval(time(16, 18), time(16, 24), "xxx", locale="ja")
'16:18:00\uff5e16:24:00'
```

```
>>> format_interval(date(2016, 1, 15), date(2016, 1, 17), "xxx", locale="de")
'15.01.2016 \u2013 17.01.2016'
```

Parameters

- `start` – First instant (datetime/date/time)
- `end` – Second instant (datetime/date/time)
- `skeleton` – The "skeleton format" to use for formatting.
- `tzinfo` – tzinfo to use (if none is already attached)
- `fuzzy` – If the skeleton is not found, allow choosing a skeleton that's close enough to it.
- `locale` – A locale object or identifier.

Returns Formatted interval

Timezone Functionality

`babel.dates.get_timezone(`*zone=None*`)`

Looks up a timezone by name and returns it. The timezone object returned comes from `pytz` and corresponds to the *tzinfo* interface and can be used with all of the functions of Babel that operate with dates.

If a timezone is not known a `LookupError` is raised. If *zone* is `None` a local zone object is returned.

Parameters `zone` – the name of the timezone to look up. If a timezone object itself is passed in, mit's returned unchanged.

`babel.dates.get_timezone_gmt(`*datetime=None*, *width='long'*, *locale=None*`)`

Return the timezone associated with the given *datetime* object formatted as string indicating the offset from GMT.

```
>>> dt = datetime(2007, 4, 1, 15, 30)
>>> get_timezone_gmt(dt, locale='en')
u'GMT+00:00'
```

```
>>> tz = get_timezone('America/Los_Angeles')
>>> dt = tz.localize(datetime(2007, 4, 1, 15, 30))
>>> get_timezone_gmt(dt, locale='en')
u'GMT-07:00'
>>> get_timezone_gmt(dt, 'short', locale='en')
u'-0700'
```

The long format depends on the locale, for example in France the acronym UTC string is used instead of GMT:

```
>>> get_timezone_gmt(dt, 'long', locale='fr_FR')
u'UTC-07:00'
```

New in version 0.9.

Parameters

- `datetime` – the `datetime` object; if *None*, the current date and time in UTC is used
- `width` – either "long" or "short"
- `locale` – the *Locale* object, or a locale string

`babel.dates.get_timezone_location(`*dt_or_tzinfo=None*, *locale=None*`)`

Return a representation of the given timezone using "location format".

The result depends on both the local display name of the country and the city associated with the time zone:

```
>>> tz = get_timezone('America/St_Johns')
>>> print(get_timezone_location(tz, locale='de_DE'))
Kanada (St. John's) Zeit
>>> tz = get_timezone('America/Mexico_City')
>>> get_timezone_location(tz, locale='de_DE')
u'Mexiko (Mexiko-Stadt) Zeit'
```

If the timezone is associated with a country that uses only a single timezone, just the localized country name is returned:

```
>>> tz = get_timezone('Europe/Berlin')
>>> get_timezone_name(tz, locale='de_DE')
u'Mitteleurop\xe4ische Zeit'
```

New in version 0.9.

Parameters

- `dt_or_tzinfo` – the `datetime` or `tzinfo` object that determines the timezone; if *None*, the current date and time in UTC is assumed
- `locale` – the *Locale* object, or a locale string

Returns the localized timezone name using location format

`babel.dates.get_timezone_name(`*dt_or_tzinfo=None*, *width='long'*, *uncommon=False*, *locale=None*, *zone_variant=None*`)`

Return the localized display name for the given timezone. The timezone may be specified using a `datetime` or *tzinfo* object.

```
>>> dt = time(15, 30, tzinfo=get_timezone('America/Los_Angeles'))
>>> get_timezone_name(dt, locale='en_US')
u'Pacific Standard Time'
```

```
>>> get_timezone_name(dt, width='short', locale='en_US')
u'PST'
```

If this function gets passed only a *tzinfo* object and no concrete *datetime*, the returned display name is indenpendent of daylight savings time. This can be used for example for selecting timezones, or to set the time of events that recur across DST changes:

```
>>> tz = get_timezone('America/Los_Angeles')
>>> get_timezone_name(tz, locale='en_US')
u'Pacific Time'
>>> get_timezone_name(tz, 'short', locale='en_US')
u'PT'
```

If no localized display name for the timezone is available, and the timezone is associated with a country that uses only a single timezone, the name of that country is returned, formatted according to the locale:

```
>>> tz = get_timezone('Europe/Berlin')
>>> get_timezone_name(tz, locale='de_DE')
u'Mitteleurop\xe4ische Zeit'
>>> get_timezone_name(tz, locale='pt_BR')
u'Hor\xe1rio da Europa Central'
```

On the other hand, if the country uses multiple timezones, the city is also included in the representation:

```
>>> tz = get_timezone('America/St_Johns')
>>> get_timezone_name(tz, locale='de_DE')
u'Neufundland-Zeit'
```

Note that short format is currently not supported for all timezones and all locales. This is partially because not every timezone has a short code in every locale. In that case it currently falls back to the long format.

For more information see LDML Appendix J: Time Zone Display Names

New in version 0.9.

Changed in version 1.0: Added *zone_variant* support.

Parameters

- `dt_or_tzinfo` – the `datetime` or `tzinfo` object that determines the timezone; if a `tzinfo` object is used, the resulting display name will be generic, i.e. independent of daylight savings time; if *None*, the current date in UTC is assumed
- `width` – either "long" or "short"
- `uncommon` – deprecated and ignored
- `zone_variant` – defines the zone variation to return. By default the variation is defined from the datetime object passed in. If no datetime object is passed in, the `'generic'` variation is assumed. The following values are valid: `'generic'`, `'daylight'` and `'standard'`.
- `locale` – the *Locale* object, or a locale string

`babel.dates.get_next_timezone_transition(`*zone=None*, *dt=None*`)`

Given a timezone it will return a `TimezoneTransition` object that holds the information about the next timezone transition that's going to happen. For instance this can be used to detect when the next DST change is going to happen and how it looks like.

The transition is calculated relative to the given datetime object. The next transition that follows the date is used. If a transition cannot be found the return value will be *None*.

Transition information can only be provided for timezones returned by the *get_timezone()* function.

Parameters

- `zone` – the timezone for which the transition should be looked up. If not provided the local timezone is used.
- `dt` – the date after which the next transition should be found. If not given the current time is assumed.

`babel.dates.UTC`
A timezone object for UTC.

`babel.dates.LOCALTZ`
A timezone object for the computer's local timezone.

Data Access

`babel.dates.get_period_names(`*locale=None*`)`
Return the names for day periods (AM/PM) used by the locale.

```
>>> get_period_names(locale='en_US')['am']
u'AM'
```

Parameters `locale` – the *Locale* object, or a locale string

`babel.dates.get_day_names(`*width='wide'*, *context='format'*, *locale=None*`)`
Return the day names used by the locale for the specified format.

```
>>> get_day_names('wide', locale='en_US')[1]
u'Tuesday'
>>> get_day_names('abbreviated', locale='es')[1]
u'mar.'
>>> get_day_names('narrow', context='stand-alone', locale='de_DE')[1]
u'D'
```

Parameters

- `width` – the width to use, one of "wide", "abbreviated", or "narrow"
- `context` – the context, either "format" or "stand-alone"
- `locale` – the *Locale* object, or a locale string

`babel.dates.get_month_names(`*width='wide'*, *context='format'*, *locale=None*`)`
Return the month names used by the locale for the specified format.

```
>>> get_month_names('wide', locale='en_US')[1]
u'January'
>>> get_month_names('abbreviated', locale='es')[1]
u'ene.'
>>> get_month_names('narrow', context='stand-alone', locale='de_DE')[1]
u'J'
```

Parameters

- `width` – the width to use, one of "wide", "abbreviated", or "narrow"
- `context` – the context, either "format" or "stand-alone"

- `locale` – the *Locale* object, or a locale string

`babel.dates.get_quarter_names(`*width='wide'*, *context='format'*, *locale=None*`)`
Return the quarter names used by the locale for the specified format.

```
>>> get_quarter_names('wide', locale='en_US')[1]
u'1st quarter'
>>> get_quarter_names('abbreviated', locale='de_DE')[1]
u'Q1'
```

Parameters

- `width` – the width to use, one of "wide", "abbreviated", or "narrow"
- `context` – the context, either "format" or "stand-alone"
- `locale` – the *Locale* object, or a locale string

`babel.dates.get_era_names(`*width='wide'*, *locale=None*`)`
Return the era names used by the locale for the specified format.

```
>>> get_era_names('wide', locale='en_US')[1]
u'Anno Domini'
>>> get_era_names('abbreviated', locale='de_DE')[1]
u'n. Chr.'
```

Parameters

- `width` – the width to use, either "wide", "abbreviated", or "narrow"
- `locale` – the *Locale* object, or a locale string

`babel.dates.get_date_format(`*format='medium'*, *locale=None*`)`
Return the date formatting patterns used by the locale for the specified format.

```
>>> get_date_format(locale='en_US')
<DateTimePattern u'MMM d, y'>
>>> get_date_format('full', locale='de_DE')
<DateTimePattern u'EEEE, d. MMMM y'>
```

Parameters

- `format` – the format to use, one of "full", "long", "medium", or "short"
- `locale` – the *Locale* object, or a locale string

`babel.dates.get_datetime_format(`*format='medium'*, *locale=None*`)`
Return the datetime formatting patterns used by the locale for the specified format.

```
>>> get_datetime_format(locale='en_US')
u'{1}, {0}'
```

Parameters

- `format` – the format to use, one of "full", "long", "medium", or "short"
- `locale` – the *Locale* object, or a locale string

babel.dates.get_time_format(*format='medium'*, *locale=None*)
Return the time formatting patterns used by the locale for the specified format.

```
>>> get_time_format(locale='en_US')
<DateTimePattern u'h:mm:ss a'>
>>> get_time_format('full', locale='de_DE')
<DateTimePattern u'HH:mm:ss zzzz'>
```

Parameters

- `format` – the format to use, one of "full", "long", "medium", or "short"
- `locale` – the *Locale* object, or a locale string

Basic Parsing

babel.dates.parse_date(*string*, *locale=None*)
Parse a date from a string.

This function uses the date format for the locale as a hint to determine the order in which the date fields appear in the string.

```
>>> parse_date('4/1/04', locale='en_US')
datetime.date(2004, 4, 1)
>>> parse_date('01.04.2004', locale='de_DE')
datetime.date(2004, 4, 1)
```

Parameters

- `string` – the string containing the date
- `locale` – a *Locale* object or a locale identifier

babel.dates.parse_time(*string*, *locale=None*)
Parse a time from a string.

This function uses the time format for the locale as a hint to determine the order in which the time fields appear in the string.

```
>>> parse_time('15:30:00', locale='en_US')
datetime.time(15, 30)
```

Parameters

- `string` – the string containing the time
- `locale` – a *Locale* object or a locale identifier

Returns the parsed time

Return type *time*

babel.dates.parse_pattern(*pattern*)
Parse date, time, and datetime format patterns.

```
>>> parse_pattern("MMMMd").format
u'%(MMMM)s%(d)s'
>>> parse_pattern("MMM d, yyyy").format
u'%(MMM)s %(d)s, %(yyyy)s'
```

Pattern can contain literal strings in single quotes:

```
>>> parse_pattern("H:mm' Uhr 'z").format
u'%(H)s:%(mm)s Uhr %(z)s'
```

An actual single quote can be used by using two adjacent single quote characters:

```
>>> parse_pattern("hh' o''clock'").format
u"%(hh)s o'clock"
```

Parameters `pattern` – the formatting pattern to parse

2.1.3 List Formatting

This module lets you format lists of items in a locale-dependent manner.

`babel.lists.format_list(`*lst*, *locale=None*`)`
Format the items in *lst* as a list.

```
>>> format_list(['apples', 'oranges', 'pears'], 'en')
u'apples, oranges, and pears'
>>> format_list(['apples', 'oranges', 'pears'], 'zh')
u'apples\u3001oranges\u548cpears'
```

Parameters

- `lst` – a sequence of items to format in to a list
- `locale` – the locale

2.1.4 Languages

The languages module provides functionality to access data about languages that is not bound to a given locale.

Official Languages

`babel.languages.get_official_languages(`*territory*, *regional=False*, *de_facto=False*`)`
Get the official language(s) for the given territory.

The language codes, if any are known, are returned in order of descending popularity.

If the *regional* flag is set, then languages which are regionally official are also returned.

If the *de_facto* flag is set, then languages which are "de facto" official are also returned.

> **Warning:** Note that the data is as up to date as the current version of the CLDR used by Babel. If you need scientifically accurate information, use another source!

Parameters

- `territory` (*str*) – Territory code
- `regional` (*bool*) – Whether to return regionally official languages too
- `de_facto` (*bool*) – Whether to return de-facto official languages too

Returns Tuple of language codes

Return type tuple[str]

babel.languages.get_territory_language_info(*territory*)

Get a dictionary of language information for a territory.

The dictionary is keyed by language code; the values are dicts with more information.

The following keys are currently known for the values:

- ***population_percent*: The percentage of the territory's population speaking the language.**
- ***official_status*: An optional string describing the officiality status of the language.** Known values are "official", "official_regional" and "de_facto_official".

Warning: Note that the data is as up to date as the current version of the CLDR used by Babel. If you need scientifically accurate information, use another source!

Note: Note that the format of the dict returned may change between Babel versions.

See http://www.unicode.org/cldr/charts/latest/supplemental/territory_language_information.html

Parameters `territory` (*str*) – Territory code

Returns Language information dictionary

Return type dict[str, dict]

2.1.5 Messages and Catalogs

Babel provides functionality to work with message catalogs. This part of the API documentation shows those parts.

Messages and Catalogs

This module provides a basic interface to hold catalog and message information. It's generally used to modify a gettext catalog but it is not being used to actually use the translations.

Catalogs

class babel.messages.catalog.Catalog(*locale=None*, *domain=None*, *header_comment=u'# Translations template for PROJECT.n# Copyright (C) YEAR ORGANIZATIONn# This file is distributed under the same license as the PROJECT project.n# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR.n#'*, *project=None*, *version=None*, *copyright_holder=None*, *msgid_bugs_address=None*, *creation_date=None*, *revision_date=None*, *last_translator=None*, *language_team=None*, *charset=None*, *fuzzy=True*)

Representation of a message catalog.

add(*id*, *string=None*, *locations=()*, *flags=()*, *auto_comments=()*, *user_comments=()*, *previous_id=()*, *lineno=None*, *context=None*)

Add or update the message with the specified ID.

```
>>> catalog = Catalog()
>>> catalog.add(u'foo')
<Message ...>
>>> catalog[u'foo']
<Message u'foo' (flags: [])>
```

This method simply constructs a *Message* object with the given arguments and invokes *__setitem__* with that object.

Parameters

- `id` – the message ID, or a `(singular, plural)` tuple for pluralizable messages
- `string` – the translated message string, or a `(singular, plural)` tuple for pluralizable messages
- `locations` – a sequence of `(filename, lineno)` tuples
- `flags` – a set or sequence of flags
- `auto_comments` – a sequence of automatic comments
- `user_comments` – a sequence of user comments
- `previous_id` – the previous message ID, or a `(singular, plural)` tuple for pluralizable messages
- `lineno` – the line number on which the msgid line was found in the PO file, if any
- `context` – the message context

`check()`

Run various validation checks on the translations in the catalog.

For every message which fails validation, this method yield a `(message, errors)` tuple, where `message` is the *Message* object and `errors` is a sequence of *TranslationError* objects.

Return type `iterator`

`delete(`*id*, *context=None*`)`

Delete the message with the specified ID and context.

Parameters

- `id` – the message ID
- `context` – the message context, or `None` for no context

`get(`*id*, *context=None*`)`

Return the message with the specified ID and context.

Parameters

- `id` – the message ID
- `context` – the message context, or `None` for no context

`header_comment`

The header comment for the catalog.

```
>>> catalog = Catalog(project='Foobar', version='1.0',
...                   copyright_holder='Foo Company')
>>> print(catalog.header_comment)
# Translations template for Foobar.
# Copyright (C) ... Foo Company
# This file is distributed under the same license as the Foobar project.
```

```
# FIRST AUTHOR <EMAIL@ADDRESS>, ....
#
```

The header can also be set from a string. Any known upper-case variables will be replaced when the header is retrieved again:

```
>>> catalog = Catalog(project='Foobar', version='1.0',
...                   copyright_holder='Foo Company')
>>> catalog.header_comment = '''\
... # The POT for my really cool PROJECT project.
... # Copyright (C) 1990-2003 ORGANIZATION
... # This file is distributed under the same license as the PROJECT
... # project.
... #'''
>>> print(catalog.header_comment)
# The POT for my really cool Foobar project.
# Copyright (C) 1990-2003 Foo Company
# This file is distributed under the same license as the Foobar
# project.
#
```

Type *unicode*

`language_team` = **None**
Name and email address of the language team.

`last_translator` = **None**
Name and email address of the last translator.

`mime_headers`
The MIME headers of the catalog, used for the special `msgid ""` entry.

The behavior of this property changes slightly depending on whether a locale is set or not, the latter indicating that the catalog is actually a template for actual translations.

Here's an example of the output for such a catalog template:

```
>>> from babel.dates import UTC
>>> created = datetime(1990, 4, 1, 15, 30, tzinfo=UTC)
>>> catalog = Catalog(project='Foobar', version='1.0',
...                   creation_date=created)
>>> for name, value in catalog.mime_headers:
...     print('%s: %s' % (name, value))
Project-Id-Version: Foobar 1.0
Report-Msgid-Bugs-To: EMAIL@ADDRESS
POT-Creation-Date: 1990-04-01 15:30+0000
PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE
Last-Translator: FULL NAME <EMAIL@ADDRESS>
Language-Team: LANGUAGE <LL@li.org>
MIME-Version: 1.0
Content-Type: text/plain; charset=utf-8
Content-Transfer-Encoding: 8bit
Generated-By: Babel ...
```

And here's an example of the output when the locale is set:

```
>>> revised = datetime(1990, 8, 3, 12, 0, tzinfo=UTC)
>>> catalog = Catalog(locale='de_DE', project='Foobar', version='1.0',
...                   creation_date=created, revision_date=revised,
```

```
...                    last_translator='John Doe <jd@example.com>',
...                    language_team='de_DE <de@example.com>')
>>> for name, value in catalog.mime_headers:
...     print('%s: %s' % (name, value))
Project-Id-Version: Foobar 1.0
Report-Msgid-Bugs-To: EMAIL@ADDRESS
POT-Creation-Date: 1990-04-01 15:30+0000
PO-Revision-Date: 1990-08-03 12:00+0000
Last-Translator: John Doe <jd@example.com>
Language: de_DE
Language-Team: de_DE <de@example.com>
Plural-Forms: nplurals=2; plural=(n != 1)
MIME-Version: 1.0
Content-Type: text/plain; charset=utf-8
Content-Transfer-Encoding: 8bit
Generated-By: Babel ...
```

Type *list*

`num_plurals`
The number of plurals used by the catalog or locale.

```
>>> Catalog(locale='en').num_plurals
2
>>> Catalog(locale='ga').num_plurals
3
```

Type *int*

`plural_expr`
The plural expression used by the catalog or locale.

```
>>> Catalog(locale='en').plural_expr
'(n != 1)'
>>> Catalog(locale='ga').plural_expr
'(n==1 ? 0 : n==2 ? 1 : 2)'
```

Type *string_types*

`plural_forms`
Return the plural forms declaration for the locale.

```
>>> Catalog(locale='en').plural_forms
'nplurals=2; plural=(n != 1)'
>>> Catalog(locale='pt_BR').plural_forms
'nplurals=2; plural=(n > 1)'
```

Type *str*

`update(`*template*, *no_fuzzy_matching=False*, *update_header_comment=False*`)`
Update the catalog based on the given template catalog.

```
>>> from babel.messages import Catalog
>>> template = Catalog()
>>> template.add('green', locations=[('main.py', 99)])
<Message ...>
```

```
>>> template.add('blue', locations=[('main.py', 100)])
<Message ...>
>>> template.add(('salad', 'salads'), locations=[('util.py', 42)])
<Message ...>
>>> catalog = Catalog(locale='de_DE')
>>> catalog.add('blue', u'blau', locations=[('main.py', 98)])
<Message ...>
>>> catalog.add('head', u'Kopf', locations=[('util.py', 33)])
<Message ...>
>>> catalog.add(('salad', 'salads'), (u'Salat', u'Salate'),
...             locations=[('util.py', 38)])
<Message ...>
```

```
>>> catalog.update(template)
>>> len(catalog)
3
```

```
>>> msg1 = catalog['green']
>>> msg1.string
>>> msg1.locations
[('main.py', 99)]
```

```
>>> msg2 = catalog['blue']
>>> msg2.string
u'blau'
>>> msg2.locations
[('main.py', 100)]
```

```
>>> msg3 = catalog['salad']
>>> msg3.string
(u'Salat', u'Salate')
>>> msg3.locations
[('util.py', 42)]
```

Messages that are in the catalog but not in the template are removed from the main collection, but can still be accessed via the *obsolete* member:

```
>>> 'head' in catalog
False
>>> list(catalog.obsolete.values())
[<Message 'head' (flags: [])>]
```

Parameters

- `template` – the reference catalog, usually read from a POT file
- `no_fuzzy_matching` – whether to use fuzzy matching of message IDs

Messages

class `babel.messages.catalog.Message`(*id*, *string=u''*, *locations=()*, *flags=()*, *auto_comments=()*, *user_comments=()*, *previous_id=()*, *lineno=None*, *context=None*)

Representation of a single message in a catalog.

`check`(*catalog=None*)

Run various validation checks on the message. Some validations are only performed if the catalog

is provided. This method returns a sequence of *TranslationError* objects.

Return type iterator

Parameters catalog – A catalog instance that is passed to the checkers

See *Catalog.check* for a way to perform checks for all messages in a catalog.

fuzzy
Whether the translation is fuzzy.

```
>>> Message('foo').fuzzy
False
>>> msg = Message('foo', 'foo', flags=['fuzzy'])
>>> msg.fuzzy
True
>>> msg
<Message 'foo' (flags: ['fuzzy'])>
```

Type *bool*

pluralizable
Whether the message is plurizable.

```
>>> Message('foo').pluralizable
False
>>> Message(('foo', 'bar')).pluralizable
True
```

Type *bool*

python_format
Whether the message contains Python-style parameters.

```
>>> Message('foo %(name)s bar').python_format
True
>>> Message(('foo %(name)s', 'foo %(name)s')).python_format
True
```

Type *bool*

Exceptions

exception babel.messages.catalog.TranslationError
Exception thrown by translation checkers when invalid message translations are encountered.

Low-Level Extraction Interface

The low level extraction interface can be used to extract from directories or files directly. Normally this is not needed as the command line tools can do that for you.

Extraction Functions

The extraction functions are what the command line tools use internally to extract strings.

babel.messages.extract.extract_from_dir(*dirname=None, method_map=[('**.py', 'python')], options_map=None, keywords={'pgettext': ((1, 'c'), 2), 'gettext': None, 'ugettext': None, 'N_': None, 'dngettext': (2, 3), 'ungettext': (1, 2), 'dgettext': (2,), '_': None, 'ngettext': (1, 2)}, comment_tags=(), callback=None, strip_comment_tags=False*)

Extract messages from any source files found in the given directory.

This function generates tuples of the form `(filename, lineno, message, comments, context)`.

Which extraction method is used per file is determined by the *method_map* parameter, which maps extended glob patterns to extraction method names. For example, the following is the default mapping:

```
>>> method_map = [
...     ('**.py', 'python')
... ]
```

This basically says that files with the filename extension ".py" at any level inside the directory should be processed by the "python" extraction method. Files that don't match any of the mapping patterns are ignored. See the documentation of the *pathmatch* function for details on the pattern syntax.

The following extended mapping would also use the "genshi" extraction method on any file in "templates" subdirectory:

```
>>> method_map = [
...     ('**/templates/**.*', 'genshi'),
...     ('**.py', 'python')
... ]
```

The dictionary provided by the optional *options_map* parameter augments these mappings. It uses extended glob patterns as keys, and the values are dictionaries mapping options names to option values (both strings).

The glob patterns of the *options_map* do not necessarily need to be the same as those used in the method mapping. For example, while all files in the `templates` folders in an application may be Genshi applications, the options for those files may differ based on extension:

```
>>> options_map = {
...     '**/templates/**.txt': {
...         'template_class': 'genshi.template:TextTemplate',
...         'encoding': 'latin-1'
...     },
...     '**/templates/**.html': {
...         'include_attrs': ''
...     }
... }
```

Parameters

- `dirname` – the path to the directory to extract messages from. If not given the current working directory is used.
- `method_map` – a list of `(pattern, method)` tuples that maps of extraction method names to extended glob patterns
- `options_map` – a dictionary of additional options (optional)
- `keywords` – a dictionary mapping keywords (i.e. names of functions that should be recognized as translation functions) to tuples that specify which of their arguments contain localizable strings

- `comment_tags` – a list of tags of translator comments to search for and include in the results
- `callback` – a function that is called for every file that message are extracted from, just before the extraction itself is performed; the function is passed the filename, the name of the extraction method and and the options dictionary as positional arguments, in that order
- `strip_comment_tags` – a flag that if set to *True* causes all comment tags to be removed from the collected comments.

See *pathmatch*

`babel.messages.extract.extract_from_file`(*method*, *filename*, *keywords={'pgettext': ((1, 'c'), 2), 'gettext': None, 'ugettext': None, 'N_': None, 'dngettext': (2, 3), 'ungettext': (1, 2), 'dgettext': (2,), '_': None, 'ngettext': (1, 2)}*, *comment_tags=()*, *options=None*, *strip_comment_tags=False*)

Extract messages from a specific file.

This function returns a list of tuples of the form `(lineno, funcname, message)`.

Parameters

- `filename` – the path to the file to extract messages from
- `method` – a string specifying the extraction method (.e.g. "python")
- `keywords` – a dictionary mapping keywords (i.e. names of functions that should be recognized as translation functions) to tuples that specify which of their arguments contain localizable strings
- `comment_tags` – a list of translator tags to search for and include in the results
- `strip_comment_tags` – a flag that if set to *True* causes all comment tags to be removed from the collected comments.
- `options` – a dictionary of additional options (optional)

`babel.messages.extract.extract`(*method*, *fileobj*, *keywords={'pgettext': ((1, 'c'), 2), 'gettext': None, 'ugettext': None, 'N_': None, 'dngettext': (2, 3), 'ungettext': (1, 2), 'dgettext': (2,), '_': None, 'ngettext': (1, 2)}*, *comment_tags=()*, *options=None*, *strip_comment_tags=False*)

Extract messages from the given file-like object using the specified extraction method.

This function returns tuples of the form `(lineno, message, comments)`.

The implementation dispatches the actual extraction to plugins, based on the value of the `method` parameter.

```
>>> source = b'''# foo module
... def run(argv):
...    print(_('Hello, world!'))
... '''
```

```
>>> from babel._compat import BytesIO
>>> for message in extract('python', BytesIO(source)):
...     print(message)
(3, u'Hello, world!', [], None)
```

Parameters

- `method` – an extraction method (a callable), or a string specifying the extraction method (.e.g. "python"); if this is a simple name, the extraction function will be looked up by entry point; if it is an explicit reference to a function (of the form `package.module:funcname` or `package.module.funcname`), the corresponding function will be imported and used
- `fileobj` – the file-like object the messages should be extracted from
- `keywords` – a dictionary mapping keywords (i.e. names of functions that should be recognized as translation functions) to tuples that specify which of their arguments contain localizable strings
- `comment_tags` – a list of translator tags to search for and include in the results
- `options` – a dictionary of additional options (optional)
- `strip_comment_tags` – a flag that if set to *True* causes all comment tags to be removed from the collected comments.

Raises `ValueError` – if the extraction method is not registered

Language Parsing

The language parsing functions are used to extract strings out of source files. These are automatically being used by the extraction functions but sometimes it can be useful to register wrapper functions, then these low level functions can be invoked.

New functions can be registered through the setuptools entrypoint system.

`babel.messages.extract.extract_python(`*fileobj*, *keywords*, *comment_ tags*, *options*`)`
Extract messages from Python source code.

It returns an iterator yielding tuples in the following form `(lineno, funcname, message, comments)`.

Parameters

- `fileobj` – the seekable, file-like object the messages should be extracted from
- `keywords` – a list of keywords (i.e. function names) that should be recognized as translation functions
- `comment_tags` – a list of translator tags to search for and include in the results
- `options` – a dictionary of additional options (optional)

Return type `iterator`

`babel.messages.extract.extract_javascript(`*fileobj*, *keywords*, *comment_ tags*, *options*`)`
Extract messages from JavaScript source code.

Parameters

- `fileobj` – the seekable, file-like object the messages should be extracted from
- `keywords` – a list of keywords (i.e. function names) that should be recognized as translation functions
- `comment_tags` – a list of translator tags to search for and include in the results
- `options` – a dictionary of additional options (optional)

`babel.messages.extract.extract_nothing(`*fileobj*, *keywords*, *comment_ tags*, *options*`)`
Pseudo extractor that does not actually extract anything, but simply returns an empty list.

MO File Support

The MO file support can read and write MO files. It reads them into *Catalog* objects and also writes catalogs out.

babel.messages.mofile.read_mo(*fileobj*)

Read a binary MO file from the given file-like object and return a corresponding *Catalog* object.

Parameters `fileobj` – the file-like object to read the MO file from

Note The implementation of this function is heavily based on the `GNUTranslations._parse` method of the `gettext` module in the standard library.

babel.messages.mofile.write_mo(*fileobj*, *catalog*, *use_fuzzy=False*)

Write a catalog to the specified file-like object using the GNU MO file format.

```
>>> import sys
>>> from babel.messages import Catalog
>>> from gettext import GNUTranslations
>>> from babel._compat import BytesIO
```

```
>>> catalog = Catalog(locale='en_US')
>>> catalog.add('foo', 'Voh')
<Message ...>
>>> catalog.add((u'bar', u'baz'), (u'Bahr', u'Batz'))
<Message ...>
>>> catalog.add('fuz', 'Futz', flags=['fuzzy'])
<Message ...>
>>> catalog.add('Fizz', '')
<Message ...>
>>> catalog.add(('Fuzz', 'Fuzzes'), ('', ''))
<Message ...>
>>> buf = BytesIO()
```

```
>>> write_mo(buf, catalog)
>>> x = buf.seek(0)
>>> translations = GNUTranslations(fp=buf)
>>> if sys.version_info[0] >= 3:
...     translations.ugettext = translations.gettext
...     translations.ungettext = translations.ngettext
>>> translations.ugettext('foo')
u'Voh'
>>> translations.ungettext('bar', 'baz', 1)
u'Bahr'
>>> translations.ungettext('bar', 'baz', 2)
u'Batz'
>>> translations.ugettext('fuz')
u'fuz'
>>> translations.ugettext('Fizz')
u'Fizz'
>>> translations.ugettext('Fuzz')
u'Fuzz'
>>> translations.ugettext('Fuzzes')
u'Fuzzes'
```

Parameters

- `fileobj` – the file-like object to write to
- `catalog` – the *Catalog* instance

- `use_fuzzy` – whether translations marked as "fuzzy" should be included in the output

PO File Support

The PO file support can read and write PO and POT files. It reads them into *Catalog* objects and also writes catalogs out.

`babel.messages.pofile.read_po(`*fileobj*, *locale=None*, *domain=None*, *ignore_obsolete=False*, *charset=None*`)`

Read messages from a `gettext` PO (portable object) file from the given file-like object and return a *Catalog*.

```
>>> from datetime import datetime
>>> from babel._compat import StringIO
>>> buf = StringIO('''
... #: main.py:1
... #, fuzzy, python-format
... msgid "foo %(name)s"
... msgstr "quux %(name)s"
...
... # A user comment
... #. An auto comment
... #: main.py:3
... msgid "bar"
... msgid_plural "baz"
... msgstr[0] "bar"
... msgstr[1] "baaz"
... ''')
>>> catalog = read_po(buf)
>>> catalog.revision_date = datetime(2007, 4, 1)
```

```
>>> for message in catalog:
...     if message.id:
...         print((message.id, message.string))
...         print(' ', (message.locations, sorted(list(message.flags))))
...         print(' ', (message.user_comments, message.auto_comments))
(u'foo %(name)s', u'quux %(name)s')
  ([(u'main.py', 1)], [u'fuzzy', u'python-format'])
  ([], [])
((u'bar', u'baz'), (u'bar', u'baaz'))
  ([(u'main.py', 3)], [])
  ([u'A user comment'], [u'An auto comment'])
```

New in version 1.0: Added support for explicit charset argument.

Parameters

- `fileobj` – the file-like object to read the PO file from
- `locale` – the locale identifier or *Locale* object, or *None* if the catalog is not bound to a locale (which basically means it's a template)
- `domain` – the message domain
- `ignore_obsolete` – whether to ignore obsolete messages in the input
- `charset` – the character set of the catalog.

`babel.messages.pofile.write_po(`*fileobj*, *catalog*, *width=76*, *no_location=False*, *omit_header=False*, *sort_output=False*, *sort_by_file=False*, *ignore_obsolete=False*, *include_previous=False*`)`

Write a `gettext` PO (portable object) template file for a given message catalog to the provided file-like object.

```
>>> catalog = Catalog()
>>> catalog.add(u'foo %(name)s', locations=[('main.py', 1)],
...             flags=('fuzzy',))
<Message...>
>>> catalog.add((u'bar', u'baz'), locations=[('main.py', 3)])
<Message...>
>>> from babel._compat import BytesIO
>>> buf = BytesIO()
>>> write_po(buf, catalog, omit_header=True)
>>> print(buf.getvalue().decode("utf8"))
#: main.py:1
#, fuzzy, python-format
msgid "foo %(name)s"
msgstr ""

#: main.py:3
msgid "bar"
msgid_plural "baz"
msgstr[0] ""
msgstr[1] ""
```

Parameters

- `fileobj` – the file-like object to write to
- `catalog` – the *Catalog* instance
- `width` – the maximum line width for the generated output; use *None*, 0, or a negative number to completely disable line wrapping
- `no_location` – do not emit a location comment for every message
- `omit_header` – do not include the `msgid ""` entry at the top of the output
- `sort_output` – whether to sort the messages in the output by msgid
- `sort_by_file` – whether to sort the messages in the output by their locations
- `ignore_obsolete` – whether to ignore obsolete messages and not include them in the output; by default they are included as comments
- `include_previous` – include the old msgid as a comment when updating the catalog

2.1.6 Numbers and Currencies

The number module provides functionality to format numbers for different locales. This includes arbitrary numbers as well as currency.

Number Formatting

`babel.numbers.format_number(`*number*, *locale=None*`)`
Return the given number formatted for a specific locale.

```
>>> format_number(1099, locale='en_US')
u'1,099'
>>> format_number(1099, locale='de_DE')
u'1.099'
```

Parameters

- `number` – the number to format
- `locale` – the *Locale* object or locale identifier

`babel.numbers.format_decimal`(*number*, *format=None*, *locale=None*)

Return the given decimal number formatted for a specific locale.

```
>>> format_decimal(1.2345, locale='en_US')
u'1.234'
>>> format_decimal(1.2346, locale='en_US')
u'1.235'
>>> format_decimal(-1.2346, locale='en_US')
u'-1.235'
>>> format_decimal(1.2345, locale='sv_SE')
u'1,234'
>>> format_decimal(1.2345, locale='de')
u'1,234'
```

The appropriate thousands grouping and the decimal separator are used for each locale:

```
>>> format_decimal(12345.5, locale='en_US')
u'12,345.5'
```

Parameters

- `number` – the number to format
- `format` –
- `locale` – the *Locale* object or locale identifier

`babel.numbers.format_currency`(*number*, *currency*, *format=None*, *locale=None*, *currency_digits=True*, *format_type='standard'*)

Return formatted currency value.

```
>>> format_currency(1099.98, 'USD', locale='en_US')
u'$1,099.98'
>>> format_currency(1099.98, 'USD', locale='es_CO')
u'US$\xa01.099,98'
>>> format_currency(1099.98, 'EUR', locale='de_DE')
u'1.099,98\xa0\u20ac'
```

The format can also be specified explicitly. The currency is placed with the '¤' sign. As the sign gets repeated the format expands (¤ being the symbol, ¤¤ is the currency abbreviation and ¤¤¤ is the full name of the currency):

```
>>> format_currency(1099.98, 'EUR', u'¤¤ #,##0.00', locale='en_US')
u'EUR 1,099.98'
>>> format_currency(1099.98, 'EUR', u'#,##0.00 ¤¤¤', locale='en_US')
u'1,099.98 euros'
```

Currencies usually have a specific number of decimal digits. This function favours that information over the given format:

```
>>> format_currency(1099.98, 'JPY', locale='en_US')
u'\xa51,100'
>>> format_currency(1099.98, 'COP', u'#,##0.00', locale='es_ES')
u'1.100'
```

However, the number of decimal digits can be overriden from the currency information, by setting the last parameter to `False`:

```
>>> format_currency(1099.98, 'JPY', locale='en_US', currency_digits=False)
u'\xa51,099.98'
>>> format_currency(1099.98, 'COP', u'#,##0.00', locale='es_ES', currency_digits=False)
u'1.099,98'
```

If a format is not specified the type of currency format to use from the locale can be specified:

```
>>> format_currency(1099.98, 'EUR', locale='en_US', format_type='standard')
u'\u20ac1,099.98'
```

When the given currency format type is not available, an exception is raised:

```
>>> format_currency('1099.98', 'EUR', locale='root', format_type='unknown')
Traceback (most recent call last):
    ...
UnknownCurrencyFormatError: "'unknown' is not a known currency format type"
```

Parameters

- `number` – the number to format
- `currency` – the currency code
- `format` – the format string to use
- `locale` – the *Locale* object or locale identifier
- `currency_digits` – use the currency's number of decimal digits
- `format_type` – the currency format type to use

`babel.numbers.format_percent(`*number*, *format=None*, *locale=None*`)`

Return formatted percent value for a specific locale.

```
>>> format_percent(0.34, locale='en_US')
u'34%'
>>> format_percent(25.1234, locale='en_US')
u'2,512%'
>>> format_percent(25.1234, locale='sv_SE')
u'2\xa0512\xa0%'
```

The format pattern can also be specified explicitly:

```
>>> format_percent(25.1234, u'#,##0\u2030', locale='en_US')
u'25,123\u2030'
```

Parameters

- `number` – the percent number to format
- `format` –
- `locale` – the *Locale* object or locale identifier

babel.numbers.format_scientific(*number*, *format=None*, *locale=None*)
Return value formatted in scientific notation for a specific locale.

```
>>> format_scientific(10000, locale='en_US')
u'1E4'
```

The format pattern can also be specified explicitly:

```
>>> format_scientific(1234567, u'##0E00', locale='en_US')
u'1.23E06'
```

Parameters

- `number` – the number to format
- `format` –
- `locale` – the *Locale* object or locale identifier

Number Parsing

babel.numbers.parse_number(*string*, *locale=None*)
Parse localized number string into an integer.

```
>>> parse_number('1,099', locale='en_US')
1099
>>> parse_number('1.099', locale='de_DE')
1099
```

When the given string cannot be parsed, an exception is raised:

```
>>> parse_number('1.099,98', locale='de')
Traceback (most recent call last):
    ...
NumberFormatError: '1.099,98' is not a valid number
```

Parameters

- `string` – the string to parse
- `locale` – the *Locale* object or locale identifier

Returns the parsed number

Raises `NumberFormatError` – if the string can not be converted to a number

babel.numbers.parse_decimal(*string*, *locale=None*)
Parse localized decimal string into a decimal.

```
>>> parse_decimal('1,099.98', locale='en_US')
Decimal('1099.98')
>>> parse_decimal('1.099,98', locale='de')
Decimal('1099.98')
```

When the given string cannot be parsed, an exception is raised:

```
>>> parse_decimal('2,109,998', locale='de')
Traceback (most recent call last):
    ...
NumberFormatError: '2,109,998' is not a valid decimal number
```

Parameters

- `string` – the string to parse
- `locale` – the *Locale* object or locale identifier

Raises `NumberFormatError` – if the string can not be converted to a decimal number

Exceptions

exception `babel.numbers.NumberFormatError`
Exception raised when a string cannot be parsed into a number.

Data Access

`babel.numbers.get_currency_name(`*currency*, *count=None*, *locale=None*`)`
Return the name used by the locale for the specified currency.

```
>>> get_currency_name('USD', locale='en_US')
u'US Dollar'
```

New in version 0.9.4.

Parameters

- `currency` – the currency code
- `count` – the optional count. If provided the currency name will be pluralized to that number if possible.
- `locale` – the *Locale* object or locale identifier

`babel.numbers.get_currency_symbol(`*currency*, *locale=None*`)`
Return the symbol used by the locale for the specified currency.

```
>>> get_currency_symbol('USD', locale='en_US')
u'$'
```

Parameters

- `currency` – the currency code
- `locale` – the *Locale* object or locale identifier

`babel.numbers.get_decimal_symbol(`*locale=None*`)`
Return the symbol used by the locale to separate decimal fractions.

```
>>> get_decimal_symbol('en_US')
u'.'
```

Parameters `locale` – the *Locale* object or locale identifier

`babel.numbers.get_plus_sign_symbol(`*locale=None*`)`
Return the plus sign symbol used by the current locale.

```
>>> get_plus_sign_symbol('en_US')
u'+'
```

Parameters `locale` – the *Locale* object or locale identifier

babel.numbers.get_minus_sign_symbol(*locale=None*)

Return the plus sign symbol used by the current locale.

```
>>> get_minus_sign_symbol('en_US')
u'-'
```

Parameters `locale` – the *Locale* object or locale identifier

babel.numbers.get_territory_currencies(*territory*, *start_date=None*, *end_date=None*, *tender=True*, *non_tender=False*, *include_details=False*)

Returns the list of currencies for the given territory that are valid for the given date range. In addition to that the currency database distinguishes between tender and non-tender currencies. By default only tender currencies are returned.

The return value is a list of all currencies roughly ordered by the time of when the currency became active. The longer the currency is being in use the more to the left of the list it will be.

The start date defaults to today. If no end date is given it will be the same as the start date. Otherwise a range can be defined. For instance this can be used to find the currencies in use in Austria between 1995 and 2011:

```
>>> from datetime import date
>>> get_territory_currencies('AT', date(1995, 1, 1), date(2011, 1, 1))
['ATS', 'EUR']
```

Likewise it's also possible to find all the currencies in use on a single date:

```
>>> get_territory_currencies('AT', date(1995, 1, 1))
['ATS']
>>> get_territory_currencies('AT', date(2011, 1, 1))
['EUR']
```

By default the return value only includes tender currencies. This however can be changed:

```
>>> get_territory_currencies('US')
['USD']
>>> get_territory_currencies('US', tender=False, non_tender=True,
...                          start_date=date(2014, 1, 1))
['USN', 'USS']
```

New in version 2.0.

Parameters

- `territory` – the name of the territory to find the currency fo
- `start_date` – the start date. If not given today is assumed.
- `end_date` – the end date. If not given the start date is assumed.
- `tender` – controls whether tender currencies should be included.
- `non_tender` – controls whether non-tender currencies should be included.
- `include_details` – if set to *True*, instead of returning currency codes the return value will be dictionaries with detail information. In that case each dictionary will have the keys `'currency'`, `'from'`, `'to'`, and `'tender'`.

2.1.7 Pluralization Support

The pluralization support provides functionality around the CLDR pluralization rules. It can parse and evaluate pluralization rules, as well as convert them to other formats such as gettext.

Basic Interface

class `babel.plural.PluralRule`(*rules*)

Represents a set of language pluralization rules. The constructor accepts a list of (tag, expr) tuples or a dict of CLDR rules. The resulting object is callable and accepts one parameter with a positive or negative number (both integer and float) for the number that indicates the plural form for a string and returns the tag for the format:

```
>>> rule = PluralRule({'one': 'n is 1'})
>>> rule(1)
'one'
>>> rule(2)
'other'
```

Currently the CLDR defines these tags: zero, one, two, few, many and other where other is an implicit default. Rules should be mutually exclusive; for a given numeric value, only one rule should apply (i.e. the condition should only be true for one of the plural rule elements.

classmethod `parse`(*rules*)

Create a *PluralRule* instance for the given rules. If the rules are a *PluralRule* object, that object is returned.

Parameters `rules` – the rules as list or dict, or a *PluralRule* object

Raises `RuleError` – if the expression is malformed

`rules`

The *PluralRule* as a dict of unicode plural rules.

```
>>> rule = PluralRule({'one': 'n is 1'})
>>> rule.rules
{'one': 'n is 1'}
```

`tags`

A set of explicitly defined tags in this rule. The implicit default `'other'` rules is not part of this set unless there is an explicit rule for it.

Conversion Functionality

`babel.plural.to_javascript`(*rule*)

Convert a list/dict of rules or a *PluralRule* object into a JavaScript function. This function depends on no external library:

```
>>> to_javascript({'one': 'n is 1'})
"(function(n) { return (n == 1) ? 'one' : 'other'; })"
```

Implementation detail: The function generated will probably evaluate expressions involved into range operations multiple times. This has the advantage that external helper functions are not required and is not a big performance hit for these simple calculations.

Parameters `rule` – the rules as list or dict, or a *PluralRule* object

Raises `RuleError` – if the expression is malformed

babel.plural.to_python(*rule*)
Convert a list/dict of rules or a *PluralRule* object into a regular Python function. This is useful in situations where you need a real function and don't are about the actual rule object:

```
>>> func = to_python({'one': 'n is 1', 'few': 'n in 2..4'})
>>> func(1)
'one'
>>> func(3)
'few'
>>> func = to_python({'one': 'n in 1,11', 'few': 'n in 3..10,13..19'})
>>> func(11)
'one'
>>> func(15)
'few'
```

Parameters `rule` – the rules as list or dict, or a *PluralRule* object

Raises `RuleError` – if the expression is malformed

babel.plural.to_gettext(*rule*)
The plural rule as gettext expression. The gettext expression is technically limited to integers and returns indices rather than tags.

```
>>> to_gettext({'one': 'n is 1', 'two': 'n is 2'})
'nplurals=3; plural=((n == 1) ? 0 : (n == 2) ? 1 : 2)'
```

Parameters `rule` – the rules as list or dict, or a *PluralRule* object

Raises `RuleError` – if the expression is malformed

2.1.8 General Support Functionality

Babel ships a few general helpers that are not being used by Babel itself but are useful in combination with functionality provided by it.

Convenience Helpers

class babel.support.Format(*locale*, *tzinfo=None*)
Wrapper class providing the various date and number formatting functions bound to a specific locale and time-zone.

```
>>> from babel.util import UTC
>>> from datetime import date
>>> fmt = Format('en_US', UTC)
>>> fmt.date(date(2007, 4, 1))
u'Apr 1, 2007'
>>> fmt.decimal(1.2345)
u'1.234'
```

currency(*number*, *currency*)
Return a number in the given currency formatted for the locale.

date(*date=None*, *format='medium'*)
Return a date formatted according to the given pattern.

```
>>> from datetime import date
>>> fmt = Format('en_US')
>>> fmt.date(date(2007, 4, 1))
u'Apr 1, 2007'
```

`datetime`(*datetime=None*, *format='medium'*)
Return a date and time formatted according to the given pattern.

```
>>> from datetime import datetime
>>> from pytz import timezone
>>> fmt = Format('en_US', tzinfo=timezone('US/Eastern'))
>>> fmt.datetime(datetime(2007, 4, 1, 15, 30))
u'Apr 1, 2007, 11:30:00 AM'
```

`decimal`(*number*, *format=None*)
Return a decimal number formatted for the locale.

```
>>> fmt = Format('en_US')
>>> fmt.decimal(1.2345)
u'1.234'
```

`number`(*number*)
Return an integer number formatted for the locale.

```
>>> fmt = Format('en_US')
>>> fmt.number(1099)
u'1,099'
```

`percent`(*number*, *format=None*)
Return a number formatted as percentage for the locale.

```
>>> fmt = Format('en_US')
>>> fmt.percent(0.34)
u'34%'
```

`scientific`(*number*)
Return a number formatted using scientific notation for the locale.

`time`(*time=None*, *format='medium'*)
Return a time formatted according to the given pattern.

```
>>> from datetime import datetime
>>> from pytz import timezone
>>> fmt = Format('en_US', tzinfo=timezone('US/Eastern'))
>>> fmt.time(datetime(2007, 4, 1, 15, 30))
u'11:30:00 AM'
```

`timedelta`(*delta*, *granularity='second'*, *threshold=0.85*, *format='medium'*, *add_direction=False*)
Return a time delta according to the rules of the given locale.

```
>>> from datetime import timedelta
>>> fmt = Format('en_US')
>>> fmt.timedelta(timedelta(weeks=11))
u'3 months'
```

class `babel.support.LazyProxy`(*func*, **args*, ***kwargs*)
Class for proxy objects that delegate to a specified function to evaluate the actual object.

```
>>> def greeting(name='world'):
...     return 'Hello, %s!' % name
```

```
>>> lazy_greeting = LazyProxy(greeting, name='Joe')
>>> print(lazy_greeting)
Hello, Joe!
>>> u'  ' + lazy_greeting
u'  Hello, Joe!'
>>> u'(%s)' % lazy_greeting
u'(Hello, Joe!)'
```

This can be used, for example, to implement lazy translation functions that delay the actual translation until the string is actually used. The rationale for such behavior is that the locale of the user may not always be available. In web applications, you only know the locale when processing a request.

The proxy implementation attempts to be as complete as possible, so that the lazy objects should mostly work as expected, for example for sorting:

```
>>> greetings = [
...     LazyProxy(greeting, 'world'),
...     LazyProxy(greeting, 'Joe'),
...     LazyProxy(greeting, 'universe'),
... ]
>>> greetings.sort()
>>> for greeting in greetings:
...     print(greeting)
Hello, Joe!
Hello, universe!
Hello, world!
```

Gettext Support

class `babel.support.Translations`(*fp=None*, *domain=None*)

An extended translation catalog class.

`add`(*translations*, *merge=True*)

Add the given translations to the catalog.

If the domain of the translations is different than that of the current catalog, they are added as a catalog that is only accessible by the various `d*gettext` functions.

Parameters

- `translations` – the *Translations* instance with the messages to add
- `merge` – whether translations for message domains that have already been added should be merged with the existing translations

classmethod `load`(*dirname=None*, *locales=None*, *domain=None*)

Load translations from the given directory.

Parameters

- `dirname` – the directory containing the `MO` files
- `locales` – the list of locales in order of preference (items in this list can be either *Locale* objects or locale strings)
- `domain` – the message domain (default: 'messages')

`merge`(*translations*)

Merge the given translations into the catalog.

Message translations in the specified catalog override any messages with the same identifier in the existing catalog.

Parameters `translations` – the *Translations* instance with the messages to merge

CHAPTER 3

Additional Notes

3.1 Babel Development

Babel as a library has a long history that goes back to the Trac project. Since then it has evolved into a independently developed project that implements data access for the CLDR project.

This document tries to explain as best as possible the general rules of the project in case you want to help out developing.

3.1.1 Tracking the CLDR

Generally the goal of the project is to work as closely as possible with the CLDR data. This has in the past caused some frustrating problems because the data is entirely out of our hand. To minimize the frustration we generally deal with CLDR updates the following way:

- bump the CLDR data only with a major release of Babel.
- never perform custom bugfixes on the CLDR data.
- never work around CLDR bugs within Babel. If you find a problem in the data, report it upstream.
- adjust the parsing of the data as soon as possible, otherwise this will spiral out of control later. This is especially the case for bigger updates that change pluralization and more.
- try not to test against specific CLDR data that is likely to change.

3.1.2 Python Versions

At the moment the following Python versions should be supported:

- Python 2.6
- Python 2.7
- Python 3.3 and up
- PyPy tracking 2.7 and 3.2 and up

While PyPy does not currently support 3.3, it does support traditional unicode literals which simplifies the entire situation tremendously.

Documentation must build on Python 2, Python 3 support for the documentation is an optional goal. Code examples in the docs preferrably are written in a style that makes them work on both 2.x and 3.x with preference to the former.

3.1.3 Unicode

Unicode is a big deal in Babel. Here is how the rules are set up:

- internally everything is unicode that makes sense to have as unicode. The exception to this rule are things which on Python 2 traditionally have been bytes. For example file names on Python 2 should be treated as bytes wherever possible.
- Encode / decode at boundaries explicitly. Never assume an encoding in a way it cannot be overridden. utf-8 should be generally considered the default encoding.
- Dot not use `unicode_literals`, instead use the `u''` string syntax. The reason for this is that the former introduces countless of unicode problems by accidentally upgrading strings to unicode which should not be. (docstrings for instance).

3.1.4 Dates and Timezones

Generally all timezone support in Babel is based on pytz which it just depends on. Babel should assume that timezone objects are pytz based because those are the only ones with an API that actually work correctly (due to the API problems with non UTC based timezones).

Assumptions to make:

- use UTC where possible.
- be super careful with local time. Do not use local time without knowing the exact timezone.
- *time* without date is a very useless construct. Do not try to support timezones for it. If you do, assume that the current local date is assumed and not utc date.

3.2 Babel Changelog

3.2.1 Version 2.3

(Feature release, release data to be decided)

3.2.2 Version 2.2

(Feature release, released on January 2nd 2016)

Bugfixes

- General: Add __hash__ to Locale. (#303) (2aa8074)
- General: Allow files with BOM if they're UTF-8 (#189) (da87edd)
- General: localedata directory is now locale-data (#109) (2d1882e)
- General: odict: Fix pop method (0a9e97e)
- General: Removed uses of datetime.date class from *.dat files (#174) (94f6830)
- Messages: Fix plural selection for chinese (531f666)
- Messages: Fix typo and add semicolon in plural_forms (5784501)

- Messages: Flatten NullTranslations.files into a list (ad11101)
- Times: FixedOffsetTimezone: fix display of negative offsets (d816803)

Features

- CLDR: Update to CLDR 28 (#292) (9f7f4d0)
- General: Add __copy__ and __deepcopy__ to LazyProxy. (a1cc3f1)
- General: Add official support for Python 3.4 and 3.5
- General: Improve odict performance by making key search O(1) (6822b7f)
- Locale: Add an ordinal_form property to Locale (#270) (b3f3430)
- Locale: Add support for list formatting (37ce4fa, be6e23d)
- Locale: Check inheritance exceptions first (3ef0d6d)
- Messages: Allow file locations without line numbers (#279) (79bc781)
- Messages: Allow passing a callable to *extract()* (#289) (3f58516)
- Messages: Support 'Language' header field of PO files (#76) (3ce842b)
- Messages: Update catalog headers from templates (e0e7ef1)
- Numbers: Properly load and expose currency format types (#201) (df676ab)
- Numbers: Use cdecimal by default when available (b6169be)
- Numbers: Use the CLDR's suggested number of decimals for format_currency (#139) (201ed50)
- Times: Add format_timedelta(format='narrow') support (edc5eb5)

3.2.3 Version 2.1

(Bugfix/minor feature release, released on September 25th 2015)

- Parse and honour the locale inheritance exceptions (https://github.com/python-babel/babel/issues/97)
- Fix Locale.parse using `global.dat` incompatible types (https://github.com/python-babel/babel/issues/174)
- Fix display of negative offsets in `FixedOffsetTimezone` (https://github.com/python-babel/babel/issues/214)
- Improved odict performance which is used during localization file build, should improve compilation time for large projects
- Add support for "narrow" format for `format_timedelta`
- Add universal wheel support
- Support 'Language' header field in .PO files (fixes https://github.com/python-babel/babel/issues/76)
- Test suite enhancements (coverage, broken tests fixed, etc)
- Documentation updated

3.2.4 Version 2.0

(Released on July 27th 2015, codename Second Coming)

- Added support for looking up currencies that belong to a territory through the *babel.numbers.get_territory_currencies()* function.
- Improved Python 3 support.
- Fixed some broken tests for timezone behavior.
- Improved various smaller things for dealing with dates.

3.2.5 Version 1.4

(bugfix release, release date to be decided)

- Fixed a bug that caused deprecated territory codes not being converted properly by the subtag resolving. This for instance showed up when trying to use `und_UK` as a language code which now properly resolves to `en_GB`.
- Fixed a bug that made it impossible to import the CLDR data from scratch on windows systems.

3.2.6 Version 1.3

(bugfix release, released on July 29th 2013)

- Fixed a bug in likely-subtag resolving for some common locales. This primarily makes `zh_CN` work again which was broken due to how it was defined in the likely subtags combined with our broken resolving. This fixes #37.
- Fixed a bug that caused pybabel to break when writing to stdout on Python 3.
- Removed a stray print that was causing issues when writing to stdout for message catalogs.

3.2.7 Version 1.2

(bugfix release, released on July 27th 2013)

- Included all tests in the tarball. Previously the include skipped past recursive folders.
- Changed how tests are invoked and added separate standalone test command. This simplifies testing of the package for linux distributors.

3.2.8 Version 1.1

(bugfix release, released on July 27th 2013)

- added dummy version requirements for pytz so that it installs on pip 1.4.
- Included tests in the tarball.

3.2.9 Version 1.0

(Released on July 26th 2013, codename Revival)

- support python 2.6, 2.7, 3.3+ and pypy - drop all other versions
- use tox for testing on different pythons
- Added support for the locale plural rules defined by the CLDR.
- Added *format_timedelta* function to support localized formatting of relative times with strings such as "2 days" or "1 month" (ticket #126).
- Fixed negative offset handling of Catalog._set_mime_headers (ticket #165).
- Fixed the case where messages containing square brackets would break with an unpack error.
- updated to CLDR 23
- Make the CLDR import script work with Python 2.7.
- Fix various typos.
- Sort output of list-locales.
- Make the POT-Creation-Date of the catalog being updated equal to POT-Creation-Date of the template used to update (ticket #148).
- Use a more explicit error message if no option or argument (command) is passed to pybabel (ticket #81).
- Keep the PO-Revision-Date if it is not the default value (ticket #148).
- Make –no-wrap work by reworking –width's default and mimic xgettext's behaviour of always wrapping comments (ticket #145).
- Add –project and –version options for commandline (ticket #173).
- Add a __ne__() method to the Local class.
- Explicitly sort instead of using sorted() and don't assume ordering (Jython compatibility).
- Removed ValueError raising for string formatting message checkers if the string does not contain any string formattings (ticket #150).
- Fix Serbian plural forms (ticket #213).
- Small speed improvement in format_date() (ticket #216).
- Fix so frontend.CommandLineInterface.run does not accumulate logging handlers (ticket #227, reported with initial patch by dfraser)
- Fix exception if environment contains an invalid locale setting (ticket #200)
- use cPickle instead of pickle for better performance (ticket #225)
- Only use bankers round algorithm as a tie breaker if there are two nearest numbers, round as usual if there is only one nearest number (ticket #267, patch by Martin)
- Allow disabling cache behaviour in LazyProxy (ticket #208, initial patch from Pedro Algarvio)
- Support for context-aware methods during message extraction (ticket #229, patch from David Rios)
- "init" and "update" commands support "–no-wrap" option (ticket #289)
- fix formatting of fraction in format_decimal() if the input value is a float with more than 7 significant digits (ticket #183)

- fix format_date() with datetime parameter (ticket #282, patch from Xavier Morel)
- fix format_decimal() with small Decimal values (ticket #214, patch from George Lund)
- fix handling of messages containing '\n' (ticket #198)
- handle irregular multi-line msgstr (no "" as first line) gracefully (ticket #171)
- parse_decimal() now returns Decimals not floats, API change (ticket #178)
- no warnings when running setup.py without installed setuptools (ticket #262)
- modified Locale.__eq__ method so Locales are only equal if all of their attributes (language, territory, script, variant) are equal
- resort to hard-coded message extractors/checkers if pkg_resources is installed but no egg-info was found (ticket #230)
- format_time() and format_datetime() now accept also floats (ticket #242)
- add babel.support.NullTranslations class similar to gettext.NullTranslations but with all of Babel's new gettext methods (ticket #277)
- "init" and "update" commands support "–width" option (ticket #284)
- fix 'input_dirs' option for setuptools integration (ticket #232, initial patch by Étienne Bersac)
- ensure .mo file header contains the same information as the source .po file (ticket #199)
- added support for get_language_name() on the locale objects.
- added support for get_territory_name() on the locale objects.
- added support for get_script_name() on the locale objects.
- added pluralization support for currency names and added a '¤¤¤' pattern for currencies that includes the full name.
- depend on pytz now and wrap it nicer. This gives us improved support for things like timezone transitions and an overall nicer API.
- Added support for explicit charset to PO file reading.
- Added experimental Python 3 support.
- Added better support for returning timezone names.
- Don't throw away a Catalog's obsolete messages when updating it.
- Added basic likelySubtag resolving when doing locale parsing and no match can be found.

3.2.10 Version 0.9.6

(released on March 17th 2011)

- Backport r493-494: documentation typo fixes.
- Make the CLDR import script work with Python 2.7.
- Fix various typos.
- Fixed Python 2.3 compatibility (ticket #146, ticket #233).
- Sort output of list-locales.
- Make the POT-Creation-Date of the catalog being updated equal to POT-Creation-Date of the template used to update (ticket #148).

- Use a more explicit error message if no option or argument (command) is passed to pybabel (ticket #81).
- Keep the PO-Revision-Date if it is not the default value (ticket #148).
- Make –no-wrap work by reworking –width's default and mimic xgettext's behaviour of always wrapping comments (ticket #145).
- Fixed negative offset handling of Catalog._set_mime_headers (ticket #165).
- Add –project and –version options for commandline (ticket #173).
- Add a __ne__() method to the Local class.
- Explicitly sort instead of using sorted() and don't assume ordering (Python 2.3 and Jython compatibility).
- Removed ValueError raising for string formatting message checkers if the string does not contain any string formattings (ticket #150).
- Fix Serbian plural forms (ticket #213).
- Small speed improvement in format_date() (ticket #216).
- Fix number formatting for locales where CLDR specifies alt or draft items (ticket #217)
- Fix bad check in format_time (ticket #257, reported with patch and tests by jomae)
- Fix so frontend.CommandLineInterface.run does not accumulate logging handlers (ticket #227, reported with initial patch by dfraser)
- Fix exception if environment contains an invalid locale setting (ticket #200)

3.2.11 Version 0.9.5

(relased on April 6th 2010)

- Fixed the case where messages containing square brackets would break with an unpack error.
- Backport of r467: Fuzzy matching regarding plurals should *NOT* be checked against len(message.id) because this is always 2, instead, it's should be checked against catalog.num_plurals (ticket #212).

3.2.12 Version 0.9.4

(released on August 25th 2008)

- Currency symbol definitions that is defined with choice patterns in the CLDR data are no longer imported, so the symbol code will be used instead.
- Fixed quarter support in date formatting.
- Fixed a serious memory leak that was introduces by the support for CLDR aliases in 0.9.3 (ticket #128).
- Locale modifiers such as "@euro" are now stripped from locale identifiers when parsing (ticket #136).
- The system locales "C" and "POSIX" are now treated as aliases for "en_US_POSIX", for which the CLDR provides the appropriate data. Thanks to Manlio Perillo for the suggestion.
- Fixed JavaScript extraction for regular expression literals (ticket #138) and concatenated strings.
- The *Translation* class in *babel.support* can now manage catalogs with different message domains, and exposes the family of *d*gettext* functions (ticket #137).

3.2.13 Version 0.9.3

(released on July 9th 2008)

- Fixed invalid message extraction methods causing an UnboundLocalError.
- Extraction method specification can now use a dot instead of the colon to separate module and function name (ticket #105).
- Fixed message catalog compilation for locales with more than two plural forms (ticket #95).
- Fixed compilation of message catalogs for locales with more than two plural forms where the translations were empty (ticket #97).
- The stripping of the comment tags in comments is optional now and is done for each line in a comment.
- Added a JavaScript message extractor.
- Updated to CLDR 1.6.
- Fixed timezone calculations when formatting datetime and time values.
- Added a *get_plural* function into the plurals module that returns the correct plural forms for a locale as tuple.
- Added support for alias definitions in the CLDR data files, meaning that the chance for items missing in certain locales should be greatly reduced (ticket #68).

3.2.14 Version 0.9.2

(released on February 4th 2008)

- Fixed catalogs' charset values not being recognized (ticket #66).
- Numerous improvements to the default plural forms.
- Fixed fuzzy matching when updating message catalogs (ticket #82).
- Fixed bug in catalog updating, that in some cases pulled in translations from different catalogs based on the same template.
- Location lines in PO files do no longer get wrapped at hyphens in file names (ticket #79).
- Fixed division by zero error in catalog compilation on empty catalogs (ticket #60).

3.2.15 Version 0.9.1

(released on September 7th 2007)

- Fixed catalog updating when a message is merged that was previously simple but now has a plural form, for example by moving from *gettext* to *ngettext*, or vice versa.
- Fixed time formatting for 12 am and 12 pm.
- Fixed output encoding of the *pybabel –list-locales* command.
- MO files are now written in binary mode on windows (ticket #61).

3.2.16 Version 0.9

(released on August 20th 2007)

- The *new_catalog* distutils command has been renamed to *init_catalog* for consistency with the command-line frontend.
- Added compilation of message catalogs to MO files (ticket #21).
- Added updating of message catalogs from POT files (ticket #22).
- Support for significant digits in number formatting.
- Apply proper "banker's rounding" in number formatting in a cross-platform manner.
- The number formatting functions now also work with numbers represented by Python *Decimal* objects (ticket #53).
- Added extensible infrastructure for validating translation catalogs.
- Fixed the extractor not filtering out messages that didn't validate against the keyword's specification (ticket #39).
- Fixed the extractor raising an exception when encountering an empty string msgid. It now emits a warning to stderr.
- Numerous Python message extractor fixes: it now handles nested function calls within a gettext function call correctly, uses the correct line number for multi-line function calls, and other small fixes (tickets ticket #38 and ticket #39).
- Improved support for detecting Python string formatting fields in message strings (ticket #57).
- CLDR upgraded to the 1.5 release.
- Improved timezone formatting.
- Implemented scientific number formatting.
- Added mechanism to lookup locales by alias, for cases where browsers insist on including only the language code in the *Accept-Language* header, and sometimes even the incorrect language code.

3.2.17 Version 0.8.1

(released on July 2nd 2007)

- *default_locale()* would fail when the value of the *LANGUAGE* environment variable contained multiple language codes separated by colon, as is explicitly allowed by the GNU gettext tools. As the *default_locale()* function is called at the module level in some modules, this bug would completely break importing these modules on systems where *LANGUAGE* is set that way.
- The character set specified in PO template files is now respected when creating new catalog files based on that template. This allows the use of characters outside the ASCII range in POT files (ticket #17).
- The default ordering of messages in generated POT files, which is based on the order those messages are found when walking the source tree, is no longer subject to differences between platforms; directory and file names are now always sorted alphabetically.
- The Python message extractor now respects the special encoding comment to be able to handle files containing non-ASCII characters (ticket #23).
- Added `N_` (gettext noop) to the extractor's default keywords.
- Made locale string parsing more robust, and also take the script part into account (ticket #27).

- Added a function to list all locales for which locale data is available.
- Added a command-line option to the *pybabel* command which prints out all available locales (ticket #24).
- The name of the command-line script has been changed from just *babel* to *pybabel* to avoid a conflict with the OpenBabel project (ticket #34).

3.2.18 Version 0.8

(released on June 20th 2007)

- First public release

3.3 License

Babel is licensed under a three clause BSD License. It basically means: do whatever you want with it as long as the copyright in Babel sticks around, the conditions are not modified and the disclaimer is present. Furthermore you must not use the names of the authors to promote derivatives of the software without written consent.

The full license text can be found below (*Babel License*).

3.3.1 Authors

Babel is written and maintained by the Babel team and various contributors:

Maintainer and Current Project Lead:

- Armin Ronacher <armin.ronacher@active-4.com>

Contributors:

- Christopher Lenz <cmlenz@gmail.com>
- Alex Morega <alex@grep.ro>
- Felix Schwarz <felix.schwarz@oss.schwarz.eu>
- Pedro Algarvio <pedro@algarvio.me>
- Jeroen Ruigrok van der Werven <asmodai@in-nomine.org>
- Philip Jenvey <pjenvey@underboss.org>
- Tobias Bieniek <Tobias.Bieniek@gmx.de>
- Jonas Borgström <jonas@edgewall.org>
- Daniel Neuhäuser <dasdasich@gmail.com>
- Nick Retallack <nick@bitcasa.com>
- Thomas Waldmann <tw@waldmann-edv.de>
- Lennart Regebro <regebro@gmail.com>
- Isaac Jurado <diptongo@gmail.com>
- Craig Loftus <craig@regulusweb.com>

Babel was previously developed under the Copyright of Edgewall Software. The following copyright notice holds true for releases before 2013: "Copyright (c) 2007 - 2011 by Edgewall Software"

In addition to the regular contributions Babel includes a fork of Lennart Regebro's tzlocal that originally was licensed under the CC0 license. The original copyright of that project is "Copyright 2013 by Lennart Regebro".

3.3.2 General License Definitions

The following section contains the full license texts for Flask and the documentation.

- "AUTHORS" hereby refers to all the authors listed in the *Authors* section.
- The "*Babel License*" applies to all the sourcecode shipped as part of Babel (Babel itself as well as the examples and the unittests) as well as documentation.

3.3.3 Babel License

Copyright (C) 2013 by the Babel Team, see AUTHORS for more information.

All rights reserved.

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. The name of the author may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR "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 AUTHOR 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.

b

S

T

U

V

W

Z