Ferenda Documentation

Release 0.2.0

Staffan Malmgren

July 23, 2014

Contents

Ferenda is a python library and framework for transforming unstructured document collections into structured Linked Data. It helps with downloading documents, parsing them to add explicit semantic structure and RDF-based metadata, finding relationships between documents, and republishing the results.

CHAPTER 1

Introduction to Ferenda

Ferenda is a python library and framework for transforming unstructured document collections into structured Linked Data. It helps with downloading documents, parsing them to add explicit semantic structure and RDF-based metadata, finding relationships between documents, and republishing the results.

It uses the XHTML and RDFa standards for representing semantic structure, and republishes content using Linked Data principles and a REST-based API.

Ferenda works best for large document collections that have some degree of internal standardization, such as the laws of a particular country, technical standards, or reports published in a series. It is particularly useful for collections that contains explicit references between documents, within or across collections.

It is designed to make it easy to get started with basic downloading, parsing and republishing of documents, and then to improve each step incrementally.

1.1 Example

Ferenda can be used either as a library or as a command-line tool. This code uses the Ferenda API to create a website containing all(*) RFCs and W3C recommended standards.

```
from ferenda.sources.tech import RFC, W3Standards
from ferenda.manager import makeresources, frontpage, runserver, setup_logger
from ferenda.errors import DocumentRemovedError, ParseError, FSMStateError

config = {'datadir':'netstandards/exampledata',
          'loglevel':'DEBUG',
          'force':False,
          'storetype':'SQLITE',
          'storelocation':'netstandards/exampledata/netstandards.sqlite',
          'storerepository':'netstandards',
          'downloadmax': 50 # remove this to download everything
}

setup_logger(level='DEBUG')

# Set up two document repositories
docrepos = (RFC(**config), W3Standards(**config))

for docrepo in docrepos:
    # Download a bunch of documents
    docrepo.download()
```

```
    # Parse all downloaded documents
    for basefile in docrepo.store.list_basefiles_for("parse"):
        try:
            docrepo.parse(basefile)
        except ParseError as e:
            pass  # or handle this in an appropriate way

    # Index the text content and metadata of all parsed documents
    for basefile in docrepo.store.list_basefiles_for("relate"):
        docrepo.relate(basefile, docrepos)

# Prepare various assets for web site navigation
makeresources(docrepos,
              resourcedir="netstandards/exampledata/rsrc",
              sitename="Netstandards",
              sitedescription="A repository of internet standard documents")

# Relate for all repos must run before generate for any repo
for docrepo in docrepos:
    # Generate static HTML files from the parsed documents,
    # with back- and forward links between them, etc.
    for basefile in docrepo.store.list_basefiles_for("generate"):
        docrepo.generate(basefile)

    # Generate a table of contents of all available documents
    docrepo.toc()
    # Generate feeds of new and updated documents, in HTML and Atom flavors
    docrepo.news()

# Create a frontpage for the entire site
frontpage(docrepos,path="netstandards/exampledata/index.html")

# Start WSGI app at http://localhost:8000/ with navigation,
# document viewing, search and API
# runserver(docrepos, port=8000, documentroot="netstandards/exampledata")
```

Alternately, using the command line tools and the project framework:

```
$ ferenda-setup netstandards
$ cd netstandards
$ ./ferenda-build.py ferenda.sources.tech.RFC enable
$ ./ferenda-build.py ferenda.sources.tech.W3Standards enable
$ ./ferenda-build.py all all --downloadmax=50
# $ ./ferenda-build.py all runserver &
# $ open http://localhost:8000/
```

Note: (*) actually, it only downloads the 50 most recent of each. Downloading, parsing, indexing and re-generating close to 7000 RFC documents takes several hours. In order to process all documents, remove the `downloadmax` configuration parameter/command line option, and be prepared to wait. You should also set up an external triple store (see *Triple stores*) and an external fulltext search engine (see *Fulltext search engines*).

1.2 Prerequisites

Operating system Ferenda is tested and works on Unix, Mac OS and Windows.

Python Version 2.6 or newer required, 3.4 recommended. The code base is primarily developed with python 3, and is heavily dependent on all forward compatibility features introduced in Python 2.6. Python 3.0 and 3.1 is not supported.

Third-party libraries `beautifulsoup4`, `rdflib`, `html5lib`, `lxml`, `requests`, `whoosh`, `pyparsing`, `jsmin`, `six` and their respective requirements. If you install ferenda using `easy_install` or `pip` they should be installed automatically. If you're working with a clone of the source repository you can install them with a simple `pip install -r requirements.py3.txt` (substitute with `requirements.py2.txt` if you're not yet using python 3).

Command-line tools For some functionality, certain executables must be present and in your `$PATH`:

- `PDFReader` requires `pdftotext` and `pdftohtml` (from poppler, version 0.21 or newer).
  - The `crop()` method requires `convert` (from ImageMagick).
  - The `convert_to_pdf` parameter to `read()` requires the `soffice` binary from either OpenOffice or LibreOffice
  - The `ocr_lang` parameter to `read()` requires `tesseract` (from tesseract-ocr), `convert` (see above) and `tiffcp` (from libtiff)
- `WordReader` requires antiword to handle old `.doc` files.
- `TripleStore` can perform some operations (bulk up- and download) much faster if curl is installed.

Once you have a large number of documents and metadata about those documents, you'll need a RDF triple store, either Sesame (at least version 2.7) or Fuseki (at least version 1.0). For document collections small enough to keep all metadata in memory you can get by with only rdflib, using either a Sqlite or a Berkely DB (aka Sleepycat/bsddb) backend. For further information, see *Triple stores*.

Similarly, once you have a large collection of text (either many short documents, or fewer long documents), you'll need an fulltext search engine to use the search feature (enabled by default). For small document collections the embedded whoosh library is used. Right now, ElasticSearch is the only supported external fulltext search engine.

As a rule of thumb, if your document collection contains over 100 000 RDF triples or 100 000 words, you should start thinking about setting up an external triple store or a fulltext search engine. See *Fulltext search engines*.

1.3 Installing

Ferenda should preferably be installed with pip (in fact, it's the only method tested):

```
pip install ferenda
```

You should definitely consider installing ferenda in a virtualenv.

Note: If you want to use the Sleepycat/bsddb backend for storing RDF data together with python 3, you need to install the `bsddb3` module. Even if you're using python 2 on Mac OS X, you might need to install this module, as the built-in `bsddb` module often has problems on this platform. It's not automatically installed by `easy_install`/`pip` as it has requirements of its own and is not essential.

On Windows, we recommend using a binary distribution of `lxml`. Unfortunately, at the time of writing, no such official distribution is for Python 3.3 or later. However, the inofficial distributions available at http://www.lfd.uci.edu/~gohlke/pythonlibs/#lxml has been tested with ferenda on python 3.3 and later, and seems to work great.

The binary distributions installs lxml into the system python library path. To make lxml available for your virtualenv, use the `--system-site-packages` command line switch when creating the virtualenv.

1.4 Features

- Handles downloading, structural parsing and regeneration of large document collections.
- Contains libraries to make reading of plain text, MS Word and PDF documents (including scanned text) as easy as HTML.
- Uses established information standards like XHTML, XSLT, XML namespaces, RDF and SPARQL as much as possible.
- Leverages your favourite python libraries: requests, beautifulsoup, rdflib, lxml, pyparsing and whoosh.
- Handle errors in upstream sources by creating one-off patch files for individiual documents.
- Easy to write reference/citation parsers and run them on document text.
- Documents in the same and other collections are automatically cross-referenced.
- Uses caches and dependency management to avoid performing the same work over and over.
- Once documents are downloaded and structured, you get a usable web site with REST API, Atom feeds and search for free.
- Web site generation can create a set of static HTML pages for offline use.

1.5 Next step

See *First steps* to set up a project and create your own simple document repository.

CHAPTER 2

First steps

Ferenda can be used in a project-like manner with a command-line tool (similar to how projects based on Django, Sphinx and Scrapy are used), or it can be used programatically through a simple API. In this guide, we'll primarily be using the command-line tool, and then show how to achieve the same thing using the API.

The first step is to create a project. Lets make a simple website that contains published standards from W3C and IETF, called "netstandards". Ferenda installs a system-wide command-line tool called `ferenda-setup` whose sole purpose is to create projects:

```
$ ferenda-setup netstandards
Prerequisites ok
Selected SQLITE as triplestore
Selected WHOOSH as search engine
Project created in netstandards
$ cd netstandards
$ ls
ferenda-build.py
ferenda.ini
wsgi.py
```

The three files created by `ferenda-setup` is another command line tool (`ferenda-build.py`) used for management of the newly created project, a WSGI application (`wsgi.py`, see *The WSGI app*) and a configuration file (`ferenda.ini`). The default configuration file specifies most, but not all, of the available configuration parameters. See *Configuration* for a full list of the standard configuration parameters.

Note: When using the API, you don't create a project or deal with configuration files in the same way. Instead, your client code is responsible for keeping track of which docrepos to use, and providing configuration when calling their methods.

2.1 Creating a Document repository class

Any document collection is handled by a *DocumentRepository* class (or *docrepo* for short), so our first task is to create a docrepo for W3C standards.

A docrepo class is responsible for downloading documents in a specific document collection. These classes can inherit from `DocumentRepository`, which amongst others provides the method `download()` for this. Since the details of how documents are made available on the web differ greatly from collection to collection, you'll often have to override the default implementation, but in this particular case, it suffices. The default implementation assumes that all documents are available from a single index page, and that the URLs of the documents follow a set pattern.

The W3C standards are set up just like that: All standards are available at `http://www.w3.org/TR/tr-status-all`. There are a lot of links to documents on that page, and not all of them are links to recommended standards. A simple way to find only the recommended standards is to see if the link follows the pattern `http://www.w3.org/TR/<year>/REC-<standardid>-<date>`.

Creating a docrepo that is able to download all web standards is then as simple as creating a subclass and setting three class properties. Create this class in the current directory (or anywhere else on your python path) and save it as `w3cstandards.py`

```
from ferenda import DocumentRepository

class W3CStandards(DocumentRepository):
    alias = "w3c"
    start_url = "http://www.w3.org/TR/tr-status-all"
    document_url_regex = "http://www.w3.org/TR/(?P<year>\d{4})/REC-(?P<basefile>.*)-(?P<date>\d+)"
```

The first property, `alias`, is required for all docrepos and controls the alias used by the command line tool for that docrepo, as well as the path where files are stored, amongst other things. If your project has a large collection of docrepos, it's important that they all have unique aliases.

The other two properties are parameters which the default implementation of `download()` uses in order to find out which documents to download. `start_url` is just a simple regular URL, while `document_url_regex` is a standard `re` regex with named groups. The group named `basefile` has special meaning, and will be used as a base for stored files and elsewhere as a short identifier for the document. For example, the web standard found at URL http://www.w3.org/TR/2012/REC-rdf-plain-literal-20121211/ will have the basefile `rdf-plain-literal`.

2.2 Using ferenda-build.py and registering docrepo classes

Next step is to enable our class. Like most tasks, this is done using the command line tool present in your project directory. To register the class (together with a short alias) in your `ferenda.ini` configuration file, run the following:

```
$ ./ferenda-build.py w3cstandards.W3CStandards enable
22:16:26 root INFO Enabled class w3cstandards.W3CStandards (alias 'w3c')
```

This creates a new section in ferenda.ini that just looks like the following:

```
[w3c]
class = w3cstandards.W3CStandards
```

From this point on, you can use the class name or the alias "w3c" interchangably:

```
$ ./ferenda-build.py w3cstandards.W3CStandards status # verbose
22:16:27 root INFO w3cstandards.W3CStandards status finished in 0.010 sec
Status for document repository 'w3c' (w3cstandards.W3CStandards)
 download: None.
 parse: None.
 generated: None.

$ ./ferenda-build.py w3c status # terse, exactly the same result
```

Note: When using the API, there is no need (nor possibility) to register docrepo classes. Your client code directly instantiates the class(es) it uses and calls methods on them.

2.3 Downloading

To test the downloading capabilities of our class, you can run the download method directly from the command line using the command line tool:

```
$ ./ferenda-build.py w3c download
22:16:31 w3c INFO Downloading max 3 documents
22:16:32 w3c INFO emotionml: downloaded from http://www.w3.org/TR/2014/REC-emotionml-20140522/
22:16:33 w3c INFO MathML3: downloaded from http://www.w3.org/TR/2014/REC-MathML3-20140410/
22:16:33 w3c INFO xml-entity-names: downloaded from http://www.w3.org/TR/2014/REC-xml-entity-names-20
# and so on...
```

After a few minutes of downloading, the result is a bunch of files in `data/w3c/downloaded`:

```
$ ls -1 data/w3c/downloaded
MathML3.html
MathML3.html.etag
emotionml.html
emotionml.html.etag
xml-entity-names.html
xml-entity-names.html.etag
```

Note: The `.etag` files are created in order to support Conditional GET, so that we don't waste our time or remote server bandwith by re-downloading documents that hasn't changed. They can be ignored and might go away in future versions of Ferenda.

We can get a overview of the status of our docrepo using the `status` command:

```
$ ./ferenda-build.py w3cstandards.W3CStandards status # verbose
22:16:27 root INFO w3cstandards.W3CStandards status finished in 0.010 sec
Status for document repository 'w3c' (w3cstandards.W3CStandards)
 download: None.
 parse: None.
 generated: None.

$ ./ferenda-build.py w3c status # terse, exactly the same result
```

Note: To do the same using the API:

```
from w3cstandards import W3CStandards
repo = W3CStandards()
repo.download()
repo.status()
# or use repo.get_status() to get all status information in a nested dict
```

Finally, if the logging information scrolls by too quickly and you want to read it again, take a look in the `data/logs` directory. Each invocation of `ferenda-build.py` creates a new log file containing the same information that is written to stdout.

2.4 Parsing

Let's try the next step in the workflow, to parse one of the documents we've downloaded.

```
$ ./ferenda-build.py w3c parse rdfa-core
22:16:45 w3c INFO rdfa-core: parse OK (4.863 sec)
22:16:45 root INFO w3c parse finished in 4.935 sec
```

By now, you might have realized that our command line tool generally is called in the following manner:

```
$ ./ferenda-build.py <docrepo> <command> [argument(s)]
```

The parse command resulted in one new file being created in `data/w3c/parsed`.

```
$ ls -1 data/w3c/parsed
rdfa-core.xhtml
```

And we can again use the `status` command to get a comprehensive overview of our document repository.

```
$ ./ferenda-build.py w3c status
22:16:47 root INFO w3c status finished in 0.032 sec
Status for document repository 'w3c' (w3cstandards.W3CStandards)
 download: xml-entity-names, rdfa-core, emotionml... (1 more)
 parse: rdfa-core. Todo: xml-entity-names, emotionml, MathML3.
 generated: None. Todo: rdfa-core.
```

Note that by default, subsequent invocations of parse won't actually parse documents that don't need parsing.

```
$ ./ferenda-build.py w3c parse rdfa-core
22:16:50 root INFO w3c parse finished in 0.019 sec
```

But during development, when you change the parsing code frequently, you'll need to override this through the `--force` flag (or set the `force` parameter in `ferenda.ini`).

```
$ ./ferenda-build.py w3c parse rdfa-core --force
22:16:56 w3c INFO rdfa-core: parse OK (5.123 sec)
22:16:56 root INFO w3c parse finished in 5.166 sec
```

Note: To do the same using the API:

```
from w3cstandards import W3CStandards
repo = W3CStandards(force=True)
repo.parse("rdfa-core")
```

Note also that you can parse all downloaded documents through the `--all` flag, and control logging verbosity by the `--loglevel` flag.

```
$ ./ferenda-build.py w3c parse --all --loglevel=DEBUG
22:16:59 w3c DEBUG xml-entity-names: Starting
22:16:59 w3c DEBUG xml-entity-names: Created data/w3c/parsed/xml-entity-names.xhtml
22:17:00 w3c DEBUG xml-entity-names: 6 triples extracted to data/w3c/distilled/xml-entity-names.rdf
22:17:00 w3c INFO xml-entity-names: parse OK (0.717 sec)
22:17:00 w3c DEBUG emotionml: Starting
22:17:00 w3c DEBUG emotionml: Created data/w3c/parsed/emotionml.xhtml
22:17:01 w3c DEBUG emotionml: 11 triples extracted to data/w3c/distilled/emotionml.rdf
22:17:01 w3c INFO emotionml: parse OK (1.174 sec)
22:17:01 w3c DEBUG MathML3: Starting
22:17:01 w3c DEBUG MathML3: Created data/w3c/parsed/MathML3.xhtml
22:17:01 w3c DEBUG MathML3: 8 triples extracted to data/w3c/distilled/MathML3.rdf
22:17:01 w3c INFO MathML3: parse OK (0.332 sec)
22:17:01 root INFO w3c parse finished in 2.247 sec
```

Note: To do the same using the API:

```
import logging
from w3cstandards import W3CStandards
# client code is responsible for setting the effective log level -- ferenda
# just emits log messages, and depends on the caller to setup the logging
# subsystem in an appropriate way
logging.getLogger().setLevel(logging.INFO)
repo = W3CStandards()
for basefile in repo.store.list_basefiles_for("parse"):
    # You you might want to try/catch the exception
    # ferenda.errors.ParseError or any of it's children here
    repo.parse(basefile)
```

Note that the API makes you explicitly list and iterate over any available files. This is so that client code has the opportunity to parallelize this work in an appropriate way.

If we take a look at the files created in `data/w3c/distilled`, we see some metadata for each document. This metadata has been automatically extracted from RDFa statements in the XHTML documents, but is so far very spartan.

Now take a look at the files created in `data/w3c/parsed`. The default implementation of `parse()` processes the DOM of the main body of the document, but some tags and attribute that are used only for formatting are stripped, such as `<style>` and `<script>`.

These documents have quite a lot of "boilerplate" text such as table of contents and links to latest and previous versions which we'd like to remove so that just the actual text is left (problem 1). And we'd like to explicitly extract some parts of the document and represent these as metadata for the document – for example the title, the publication date, the authors/editors of the document and it's abstract, if available (problem 2).

Just like the default implementation of `download()` allowed for some customization using class variables, we can solve problem 1 by setting two additional class variables:

```
    parse_content_selector="body"
    parse_filter_selectors=["div.toc", "div.head"]
```

The `parse_content_selector` member specifies, using CSS selector syntax, the part of the document which contains our main text. It defaults to `"body"`, and can often be set to `".content"` (the first element that has a class="content" attribute"), `"#main-text"` (any element with the id `"main-text"`), `"article"` (the first `<article>` element) or similar. The `parse_filter_selectors` is a list of similar selectors, with the difference that all matching elements are removed from the tree. In this case, we use it to remove some boilerplate sections that often within the content specified by `parse_content_selector`, but which we don't want to appear in the final result.

In order to solve problem 2, we can override one of the methods that the default implementation of parse() calls:

```
    def parse_metadata_from_soup(self, soup, doc):
        from rdflib import Namespace
        from ferenda import Describer
        from ferenda import util
        import re
        DCTERMS = Namespace("http://purl.org/dc/terms/")
        FOAF = Namespace("http://xmlns.com/foaf/0.1/")
        d = Describer(doc.meta, doc.uri)
        d.rdftype(FOAF.Document)
        d.value(DCTERMS.title, soup.find("title").text, lang=doc.lang)
        d.value(DCTERMS.abstract, soup.find(True, "abstract"), lang=doc.lang)
        # find the issued date -- assume it's the first thing that looks
        # like a date on the form "22 August 2013"
        re_date = re.compile(r'(\d+ \w+ \d{4})')
        datenode = soup.find(text=re_date)
```

```
        datestr = re_date.search(datenode).group(1)
        d.value(DCTERMS.issued, util.strptime(datestr, "%d %B %Y"))
        editors = soup.find("dt", text=re.compile("Editors?:"))
        for editor in editors.find_next_siblings("dd"):
            editor_name = editor.text.strip().split(", ")[0]
            d.value(DCTERMS.editor, editor_name)
```

`parse_metadata_from_soup()` is called with a document object and the parsed HTML document in the form of a BeautifulSoup object. It is the responsibility of `parse_metadata_from_soup()` to add document-level metadata for this document, such as it's title, publication date, and similar. Note that `parse_metadata_from_soup()` is run before the `parse_content_selector` and `parse_filter_selectors` are applied, so the BeautifulSoup object passed into it contains the entire document.

Note: The selectors are passed to BeautifulSoup.select(), which supports a subset of the CSS selector syntax. If you stick with simple tag, id and class-based selectors you should be fine.

Now, if you run `parse --force` again, both documents and metadata are in better shape. Further down the line the value of properly extracted metadata will become more obvious.

2.5 Republishing the parsed content

The XHTML contains metadata in RDFa format. As such, you can extract all that metadata and put it into a triple store. The relate command does this, as well as creating a full text index of all textual content:

```
$ ./ferenda-build.py w3c relate --all
22:17:02 w3c INFO Clearing context http://localhost:8000/dataset/w3c at repository ferenda
22:17:03 w3c INFO xml-entity-names: relate OK (0.618 sec)
22:17:04 w3c INFO rdfa-core: relate OK (1.542 sec)
22:17:06 w3c INFO emotionml: relate OK (1.647 sec)
22:17:08 w3c INFO MathML3: relate OK (1.604 sec)
22:17:08 w3c INFO Dumped 34 triples from context http://localhost:8000/dataset/w3c to data/w3c/distil
22:17:08 root INFO w3c relate finished in 5.555 sec
```

The next step is to create a number of *resource files* (placed under `data/rsrc`). These resource files include css and javascript files for the new website we're creating, as well as a xml configuration file used by the XSLT transformation done by `generate` below:

```
$ ./ferenda-build.py w3c makeresources
22:17:08 root INFO Wrote data/rsrc/resources.xml
$ find data/rsrc -print
data/rsrc
data/rsrc/api
data/rsrc/api/common.json
data/rsrc/api/context.json
data/rsrc/api/terms.json
data/rsrc/css
data/rsrc/css/ferenda.css
data/rsrc/css/main.css
data/rsrc/css/normalize-1.1.3.css
data/rsrc/img
data/rsrc/img/navmenu-small-black.png
data/rsrc/img/navmenu.png
data/rsrc/img/search.png
data/rsrc/js
data/rsrc/js/ferenda.js
```

```
data/rsrc/js/jquery-1.10.2.js
data/rsrc/js/modernizr-2.6.3.js
data/rsrc/js/respond-1.3.0.js
data/rsrc/resources.xml
```

Note: It is possible to combine and minify both javascript and css files using the `combineresources` option in the configuration file.

Running `makeresources` is needed for the final few steps.

```
$ ./ferenda-build.py w3c generate --all
22:17:14 w3c INFO xml-entity-names: generate OK (1.728 sec)
22:17:14 w3c INFO rdfa-core: generate OK (0.242 sec)
22:17:14 w3c INFO emotionml: generate OK (0.336 sec)
22:17:14 w3c INFO MathML3: generate OK (0.216 sec)
22:17:14 root INFO w3c generate finished in 2.535 sec
```

The `generate` command creates browser-ready HTML5 documents from our structured XHTML documents, using our site's navigation.

```
$ ./ferenda-build.py w3c toc
22:17:17 w3c INFO Created data/w3c/toc/dcterms_issued/2013.html
22:17:17 w3c INFO Created data/w3c/toc/dcterms_issued/2014.html
22:17:17 w3c INFO Created data/w3c/toc/dcterms_title/e.html
22:17:17 w3c INFO Created data/w3c/toc/dcterms_title/m.html
22:17:17 w3c INFO Created data/w3c/toc/dcterms_title/r.html
22:17:17 w3c INFO Created data/w3c/toc/dcterms_title/x.html
22:17:18 w3c INFO Created data/w3c/toc/index.html
22:17:18 root INFO w3c toc finished in 2.059 sec
$ ./ferenda-build.py w3c news
22:17:19 w3c INFO feed main: 4 entries
22:17:19 root INFO w3c news finished in 0.115 sec
$ ./ferenda-build.py w3c frontpage
22:17:21 root INFO frontpage: wrote data/index.html (0.112 sec)
```

The `toc` and `feeds` commands creates static files for general indexes/tables of contents of all documents in our docrepo as well as Atom feeds, and the `frontpage` command creates a suitable frontpage for the site as a whole.

Note: To do all of the above using the API:

```
from ferenda import manager
from w3cstandards import W3CStandards
repo = W3CStandards()
for basefile in repo.store.list_basefiles_for("relate"):
    repo.relate(basefile)
manager.makeresources([repo], sitename="Standards", sitedescription="W3C standards, in a new form")
for basefile in repo.store.list_basefiles_for("generate"):
    repo.generate(basefile)
repo.toc()
repo.news()
manager.frontpage([repo])
```

Finally, to start a development web server and check out the finished result:

```
$ ./ferenda-build.py w3c runserver
$ open http://localhost:8080/
```

Now you've created your own web site with structured documents. It contains listings of all documents, feeds with

updated documents (in both HTML and Atom flavors), full text search, and an API. In order to deploy your site, you can run it under Apache+mod_wsgi, ngnix+uWSGI, Gunicorn or just about any WSGI capable web server, see *The WSGI app*.

Note: Using `runserver()` from the API does not really make any sense. If your environment supports running WSGI applications, see the above link for information about how to get the ferenda WSGI application. Otherwise, the app can be run by any standard WSGI host.

To keep it up-to-date whenever the W3C issues new standards, use the following command:

```
$ ./ferenda-build.py w3c all
22:17:25 w3c INFO Downloading max 3 documents
22:17:25 root INFO w3cstandards.W3CStandards download finished in 2.648 sec
22:17:25 root INFO w3cstandards.W3CStandards parse finished in 0.019 sec
22:17:25 root INFO w3cstandards.W3CStandards relate: Nothing to do!
22:17:25 root INFO w3cstandards.W3CStandards relate finished in 0.025 sec
22:17:25 root INFO Wrote data/rsrc/resources.xml
22:17:29 w3c INFO xml-entity-names: generate OK (0.000 sec)
22:17:29 w3c INFO rdfa-core: generate OK (0.000 sec)
22:17:29 w3c INFO emotionml: generate OK (0.000 sec)
22:17:29 w3c INFO MathML3: generate OK (0.000 sec)
22:17:29 root INFO w3cstandards.W3CStandards generate finished in 0.006 sec
22:17:32 w3c INFO Created data/w3c/toc/dcterms_issued/2013.html
22:17:32 w3c INFO Created data/w3c/toc/dcterms_issued/2014.html
22:17:32 w3c INFO Created data/w3c/toc/dcterms_title/e.html
22:17:32 w3c INFO Created data/w3c/toc/dcterms_title/m.html
22:17:32 w3c INFO Created data/w3c/toc/dcterms_title/r.html
22:17:32 w3c INFO Created data/w3c/toc/dcterms_title/x.html
22:17:32 w3c INFO Created data/w3c/toc/index.html
22:17:32 root INFO w3cstandards.W3CStandards toc finished in 3.376 sec
22:17:32 w3c INFO feed main: 4 entries
22:17:32 root INFO w3cstandards.W3CStandards news finished in 0.063 sec
22:17:32 root INFO frontpage: wrote data/index.html (0.017 sec)
```

The "all" command is an alias that runs `download`, `parse --all`, `relate --all`, `generate --all`, `toc` and `feeds` in sequence.

Note: The API doesn't have any corresponding method. Just run all of the above code again. As long as you don't pass the `force=True` parameter when creating the docrepo instance, ferendas dependency management should make sure that documents aren't needlessly re-parsed etc.

This 20-line example of a docrepo took a lot of shortcuts by depending on the default implementation of the `download()` and `parse()` methods. Ferenda tries to make it really to get *something* up and running quickly, and then improving each step incrementally.

In the next section *Creating your own document repositories* we will take a closer look at each of the six main steps (`download`, `parse`, `relate`, `generate`, `toc` and `news`), including how to completely replace the built-in methods. You can also take a look at the source code for `ferenda.sources.tech.W3Standards`, which contains a more complete (and substantially longer) implementation of `download()`, `parse()` and the others.

CHAPTER 3

Creating your own document repositories

The next step is to do more substantial adjustments to the download/parse/generate cycle. As the source for our next docrepo we'll use the collected RFCs, as published by IETF. These documents are mainly available in plain text format (formatted for printing on a line printer), as is the document index itself. This means that we cannot rely on the default implementation of download and parse. Furthermore, RFCs are categorized and refer to each other using varying semantics. This metadata can be captured, queried and used in a number of ways to present the RFC collection in a better way.

3.1 Writing your own `download` implementation

The purpose of the `download()` method is to fetch source documents from a remote source and store them locally, possibly under different filenames but otherwise bit-for-bit identical with how they were stored at the remote source (see *File storage* for more information about how and where files are stored locally).

The default implementation of `download()` uses a small number of methods and class variables to do the actual work. By selectively overriding these, you can often avoid rewriting a complete implementation of `download()`.

3.1.1 A simple example

We'll start out by creating a class similar to our W3C class in *First steps*. All RFC documents are listed in the index file at http://www.ietf.org/download/rfc-index.txt, while a individual document (such as RFC 6725) are available at http://tools.ietf.org/rfc/rfc6725.txt. Our first attempt will look like this (save as `rfcs.py`)

```
import re
from datetime import datetime, date

import requests

from ferenda import DocumentRepository, TextReader
from ferenda import util
from ferenda.decorators import downloadmax

class RFCs(DocumentRepository):
    alias = "rfc"
    start_url = "http://www.ietf.org/download/rfc-index.txt"
    document_url_template = "http://tools.ietf.org/rfc/rfc%(basefile)s.txt"
    downloaded_suffix = ".txt"
```

And we'll enable it and try to run it like before:

```
$ ./ferenda-build.py rfcs.RFCs enable
$ ./ferenda-build.py rfc download
```

This doesn't work! This is because start page contains no actual HTML links – it's a plaintext file. We need to parse the index text file to find out all available basefiles. In order to do that, we must override `download()`.

```
    def download(self):
        self.log.debug("download: Start at %s" %  self.start_url)
        indextext = requests.get(self.start_url).text
        reader = TextReader(string=indextext)  # see TextReader class
        iterator = reader.getiterator(reader.readparagraph)
        if not isinstance(self.config.downloadmax, (int, type(None))):
            self.config.downloadmax = int(self.config.downloadmax)

        for basefile in self.download_get_basefiles(iterator):
            self.download_single(basefile)

    @downloadmax
    def download_get_basefiles(self, source):
        for p in reversed(list(source)):
            if re.match("^(\d{4}) ",p): # looks like a RFC number
                if not "Not Issued." in p: # Skip RFC known to not exist
                    basefile = str(int(p[:4]))  # eg. '0822' -> '822'
                    yield basefile
```

Since the RFC index is a plain text file, we use the `TextReader` class, which contains a bunch of functionality to make it easier to work with plain text files. In this case, we'll iterate through the file one paragraph at a time, and if the paragraph starts with a four-digit number (and the number hasn't been marked "Not Issued.") we'll download it by calling `download_single()`.

Like the default implementation, we offload the main work to `download_single()`, which will look if the file exists on disk and only if not, attempt to download it. If the `--refresh` parameter is provided, a conditional get is performed and only if the server says the document has changed, it is re-downloaded.

Note: In many cases, the URL for the downloaded document is not easily constructed from a basefile identifier. `download_single()` therefore takes a optional url argument. The above could be written more verbosely like:

```
url = "http://tools.ietf.org/rfc/rfc%s.txt" % basefile
self.download_single(basefile, url)
```

In other cases, a document to be downloaded could consists of several resources (eg. a HTML document with images, or a PDF document with the actual content combined with a HTML document with document metadata). For these cases, you need to override `download_single()`.

3.1.2 The main flow of the download process

The main flow is that the `download()` method itself does some source-specific setup, which often include downloading some sort of index or search results page. The location of that index resource is given by the class variable `start_url`. `download()` then calls `download_get_basefiles()` which returns an iterator of basefiles.

For each basefile, `download_single()` is called. This method is responsible for downloading everything related to a single document. Most of the time, this is just a single file, but can occasionally be a set of files (like a HTML document with accompanying images, or a set of PDF files that conceptually is a single document).

The default implementation of `download_single()` assumes that a document is just a single file, and calculates the URL of that document by calling the `remote_url()` method.

The default `remote_url()` method uses the class variable `document_url_template`. This string template should be using string formatting and expect a variable called `basefile`. The default implementation of `remote_url()` can in other words only be used if the URLs of the remote source are predictable and directly based on the `basefile`.

Note: In many cases, the URL for the remote version of a document can be impossible to calculate from the basefile only, but be readily available from the main index page or search result page. For those cases, `download_get_basefiles()` should return a iterator that yields `(basefile, url)` tuples. The default implementation of `download()` handles this and uses `url` as the second, optional argument to download_single.

Finally, the actual downloading of individual files is done by the `download_if_needed()` method. As the name implies, this method tries to avoid downloading anything from the network if it's not strictly needed. If there is a file in-place already, a conditional GET is done (using the timestamp of the file for a `If-modified-since` header, and an associated .etag file for a `If-none-match` header). This avoids re-downloading the (potentially large) file if it hasn't changed.

To summarize: The main chain of calls looks something like this:

```
download
  start_url (class variable)
  download_get_basefiles (instancemethod) - iterator
  download_single (instancemethod)
     remote_url (instancemethod)
         document_url_template (class variable)
     download_if_needed (instancemethod)
```

These are the methods that you may override, and when you might want to do so:

method	Default behaviour	Override when
download	Download the contents of `start_url` and extracts all links by `lxml.html.iterlinks`, which are passed to `download_get_basefiles`. For each item that is returned, call download_single.	All your documents are not linked from a single index page (i.e. paged search results). In these cases, you should override `download_get_basefiles` as well and make that method responsible for fetching all pages of search results.
download_get_basefiles	Iterate through the (element, attribute, link, url) tuples from the source and examine if link matches `basefile_regex` or if url match `document_url_regex`. If so, yield a (text, url) tuple.	The basefile/url extraction is more complicated than what can be achieved through the `basefile_regex` / `document_url_regex` mechanism, or when you've overridden download to pass a different argument than a link iterator. Note that you must return an iterator by using the `yield` statement for each basefile found.
download_single	Calculates the url of the document to download (or, if a URL is provided, uses that), and calls `download_if_needed` with that. Afterwards, updates the `DocumentEntry` of the document to reflect source url and download timestamps.	The complete contents of your document is contained in several different files. In these cases, you should start with the main one and call `download_if_needed` for that, then calculate urls and file paths (using the `attachment` parameter to `store.downloaded_path`) for each additional file, then call `download_if_needed` for each. Finally, you must update the `DocumentEntry` object.
remote_url	Calculates a URL from a basename using `document_url_template`	The rules for producing a URL from a basefile is more complicated than what string formatting can achieve.
download_if_needed	Downloads an individual URL to a local file. Makes sure the local file has the same timestamp as the Last-modified header from the server. If an older version of the file is present, this can either be archived (the default) or overwritten.	You really shouldn't.

3.1.3 The optional basefile argument

During early stages of development, it's often useful to just download a single document, both in order to check out that download_single works as it should, and to have sample documents for parse. When using the ferenda-build.py tool, the download command can take a single optional parameter, ie.:

```
./ferenda-build.py rfc download 6725
```

If provided, this parameter is passed to the download method as the optional basefile parameter. The default implementation of download checks if this parameter is provided, and if so, simply calls download_single with that parameter, skipping the full download procedure. If you're overriding download, you should support this usage, by starting your implementation with something like this:

```
def download(self, basefile=None):
    if basefile:
        return self.download_single(basefile)

    # the rest of your code
```

3.1.4 The `downloadmax()` decorator

As we saw in *Introduction to Ferenda*, the built-in docrepos support a `downloadmax` configuration parameter. The effect of this parameter is simply to interrupt the downloading process after a certain amount of documents have been downloaded. This can be useful when doing integration-type testing, or if you just want to make it easy for someone else to try out your docrepo class. The separation between the main `download()` method anbd the `download_get_basefiles()` helper method makes this easy – just add the `@downloadmax()` to the latter. This decorator reads the `downloadmax` configuration parameter (it also looks for a `FERENDA_DOWNLOADMAX` environment variable) and if set, limits the number of basefiles returned by `download_get_basefiles()`.

3.2 Writing your own `parse` implementation

The purpose of the `parse()` method is to take the downloaded file(s) for a particular document and parse it into a structured document with proper metadata, both for the document as a whole, but also for individual sections of the document.

```
    # In order to properly handle our RDF data, we need to tell
    # ferenda which namespaces we'll be using. These will be available
    # as rdflib.Namespace objects in the self.ns dict, which means you
    # can state that something is eg. a dcterms:title by using
    # self.ns['dcterms'].title. See
    # :py:data:`~ferenda.DocumentRepository.namespaces`
    namespaces = ('rdf',  # always needed
                  'dcterms',  # title, identifier, etc
                  'bibo', # Standard and DocumentPart classes, chapter prop
                  'xsd',  # datatypes
                  'foaf', # rfcs are foaf:Documents for now
                  ('rfc','http://example.org/ontology/rfc/')
                  )
    from rdflib import Namespace
    rdf_type = Namespace('http://example.org/ontology/rfc/').RFC


    from ferenda.decorators import managedparsing

    @managedparsing
    def parse(self, doc):
        # some very simple heuristic rules for determining
        # what an individual paragraph is

        def is_heading(p):
            # If it's on a single line and it isn't indented with spaces
            # it's probably a heading.
            if p.count("\n") == 0 and not p.startswith(" "):
                return True

        def is_pagebreak(p):
            # if it contains a form feed character, it represents a page break
            return "\f" in p

        # Parsing a document consists mainly of two parts:
        # 1: First we parse the body of text and store it in doc.body
        from ferenda.elements import Body, Preformatted, Title, Heading
        from ferenda import Describer
        reader = TextReader(self.store.downloaded_path(doc.basefile))
```

```
        # First paragraph of an RFC is always a header block
        header = reader.readparagraph()
        # Preformatted is a ferenda.elements class representing a
        # block of preformatted text. It is derived from the built-in
        # list type, and must thus be initialized with an iterable, in
        # this case a single-element list of strings. (Note: if you
        # try to initialize it with a string, because strings are
        # iterables as well, you'll end up with a list where each
        # character in the string is an element, which is not what you
        # want).
        preheader = Preformatted([header])
        # Doc.body is a ferenda.elements.Body class, which is also
        # is derived from list, so it has (amongst others) the append
        # method. We build our document by adding to this root
        # element.
        doc.body.append(preheader)

        # Second paragraph is always the title, and we don't include
        # this in the body of the document, since we'll add it to the
        # medata -- once is enough
        title = reader.readparagraph()

        # After that, just iterate over the document and guess what
        # everything is. TextReader.getiterator is useful for
        # iterating through a text in other chunks than single lines
        for para in reader.getiterator(reader.readparagraph):
            if is_heading(para):
                # Heading is yet another of these ferenda.elements
                # classes.
                doc.body.append(Heading([para]))
            elif is_pagebreak(para):
                # Just drop these remnants of a page-and-paper-based past
                pass
            else:
                # If we don't know that it's something else, it's a
                # preformatted section (the safest bet for RFC text).
                doc.body.append(Preformatted([para]))

        # 2: Then we create metadata for the document and store it in
        # doc.meta (in this case using the convenience
        # ferenda.Describer class).

        desc = Describer(doc.meta, doc.uri)

        # Set the rdf:type of the document
        desc.rdftype(self.rdf_type)

        # Set the title we've captured as the dcterms:title of the document and
        # specify that it is in English
        desc.value(self.ns['dcterms'].title, util.normalize_space(title), lang="en")

        # Construct the dcterms:identifier (eg "RFC 6991") for this document from the basefile
        desc.value(self.ns['dcterms'].identifier, "RFC " + doc.basefile)

        # find and convert the publication date in the header to a datetime
        # object, and set it as the dcterms:issued date for the document
        re_date = re.compile("(January|February|March|April|May|June|July|August|September|October|No
        # This is a context manager that temporarily sets the system
```

```
        # locale to the "C" locale in order to be able to use strptime
        # with a string on the form "August 2013", even though the
        # system may use another locale.
        dt_match = re_date(header)
        if dt_match:
            with util.c_locale():
                dt = datetime.strptime(re_date(header).group(0), "%B %Y")
            pubdate = date(dt.year,dt.month,dt.day)
            # Note that using some python types (cf. datetime.date)
            # results in a datatyped RDF literal, ie in this case
            #   <http://localhost:8000/res/rfc/6994> dcterms:issued "2013-08-01"^^xsd:date
            desc.value(self.ns['dcterms'].issued, pubdate)

        # find any older RFCs that this document updates or obsoletes
        obsoletes = re.search("^Obsoletes: ([\d+, ]+)", header, re.MULTILINE)
        updates = re.search("^Updates: ([\d+, ]+)", header, re.MULTILINE)

        # Find the category of this RFC, store it as dcterms:subject
        cat_match = re.search("^Category: ([\w ]+?)(  |$)", header, re.MULTILINE)
        if cat_match:
            desc.value(self.ns['dcterms'].subject, cat_match.group(1))

        for predicate, matches in ((self.ns['rfc'].updates, updates),
                                   (self.ns['rfc'].obsoletes, obsoletes)):
            if matches is None:
                continue
            # add references between this document and these older rfcs,
            # using either rfc:updates or rfc:obsoletes
            for match in matches.group(1).strip().split(", "):
                uri = self.canonical_uri(match)
                # Note that this uses our own unofficial
                # namespace/vocabulary
                # http://example.org/ontology/rfc/
                desc.rel(predicate, uri)

        # And now we're done. We don't need to return anything as
        # we've modified the Document object that was passed to
        # us. The calling code will serialize this modified object to
        # XHTML and RDF and store it on disk
```

This implementation builds a very simple object model of a RFC document, which is serialized to a XHTML1.1+RDFa document by the `managedparsing()` decorator. If you run it (by calling `ferenda-build.py rfc parse --all`) after having downloaded the rfc documents, the result will be a set of documents in `data/rfc/parsed`, and a set of RDF files in `data/rfc/distilled`. Take a look at them! The above might appear to be a lot of code, but it also accomplishes much. Furthermore, it should be obvious how to extend it, for instance to create more metadata from the fields in the header (such as capturing the RFC category, the publishing party, the authors etc) and better semantic representation of the body (such as marking up regular paragraphs, line drawings, bulleted lists, definition lists, EBNF definitions and so on).

Next up, we'll extend this implementation in two ways: First by representing the nested nature of the sections and subsections in the documents, secondly by finding and linking citations/references to other parts of the text or other RFCs in full.

Note: How does `./ferenda-build.py rfc parse --all` work? It calls `list_basefiles_for()` with the argument `parse`, which lists all downloaded files, and extracts the basefile for each of them, then calls parse for each in turn.

3.2.1 Handling document structure

The main text of a RFC is structured into sections, which may contain subsections, which in turn can contain subsubsections. The start of each section is easy to identify, which means we can build a model of this structure by extending our parse method with relatively few lines:

```
from ferenda.elements import Section, Subsection, Subsubsection

# More heuristic rules: Section headers start at the beginning
# of a line and are numbered. Subsections and subsubsections
# have dotted numbers, optionally with a trailing period, ie
# '9.2.' or '11.3.1'
def is_section(p):
    return re.match(r"\d+\.? +[A-Z]", p)

def is_subsection(p):
    return re.match(r"\d+\.\d+\.? +[A-Z]", p)

def is_subsubsection(p):
    return re.match(r"\d+\.\d+\.\d+\.? +[A-Z]", p)

def split_sectionheader(p):
    # returns a tuple of title, ordinal, identifier
    ordinal, title = p.split(" ",1)
    ordinal = ordinal.strip(".")
    return title.strip(), ordinal, "RFC %s, section %s" % (doc.basefile, ordinal)

# Use a list as a simple stack to keep track of the nesting
# depth of a document. Every time we create a Section,
# Subsection or Subsubsection object, we push it onto the
# stack (and clear the stack down to the appropriate nesting
# depth). Every time we create some other object, we append it
# to whatever object is at the top of the stack. As your rules
# for representing the nesting of structure become more
# complicated, you might want to use the
# :class:`~ferenda.FSMParser` class, which lets you define
# heuristic rules (recognizers), states and transitions, and
# takes care of putting your structure together.
stack = [doc.body]

for para in reader.getiterator(reader.readparagraph):
    if is_section(para):
        title, ordinal, identifier = split_sectionheader(para)
        s = Section(title=title, ordinal=ordinal, identifier=identifier)
        stack[1:] = [] # clear all but bottom element
        stack[0].append(s) # add new section to body
        stack.append(s)    # push new section on top of stack
    elif is_subsection(para):
        title, ordinal, identifier = split_sectionheader(para)
        s = Subsection(title=title, ordinal=ordinal, identifier=identifier)
        stack[2:] = [] # clear all but bottom two elements
        stack[1].append(s) # add new subsection to current section
        stack.append(s)
    elif is_subsubsection(para):
        title, ordinal, identifier = split_sectionheader(para)
        s = Subsubsection(title=title, ordinal=ordinal, identifier=identifier)
        stack[3:] = [] # clear all but bottom three
        stack[-1].append(s) # add new subsubsection to current subsection
```

```
            stack.append(s)
        elif is_heading(para):
            stack[-1].append(Heading([para]))
        elif is_pagebreak(para):
            pass
        else:
            pre = Preformatted([para])
            stack[-1].append(pre)
```

This enhances parse so that instead of outputting a single long list of elements directly under `body`:

```
<h1>2.  Overview</h1>
<h1>2.1.  Date, Location, and Participants</h1>
<pre>
   The second ForCES interoperability test meeting was held by the IETF
   ForCES Working Group on February 24-25, 2011...
</pre>
<h1>2.2.  Testbed Configuration</h1>
<h1>2.2.1.  Participants' Access</h1>
<pre>
   NTT and ZJSU were physically present for the testing at the Internet
  Technology Lab (ITL) at Zhejiang Gongshang University in China.
</pre>
```

...we have a properly nested element structure, as well as much more metadata represented in RDFa form:

```
<div class="section" property="dcterms:title" content=" Overview"
     typeof="bibo:DocumentPart" about="http://localhost:8000/res/rfc/6984#S2.">
  <span property="bibo:chapter" content="2."
        about="http://localhost:8000/res/rfc/6984#S2."/>
  <div class="subsection" property="dcterms:title" content=" Date, Location, and Participants"
       typeof="bibo:DocumentPart" about="http://localhost:8000/res/rfc/6984#S2.1.">
    <span property="bibo:chapter" content="2.1."
          about="http://localhost:8000/res/rfc/6984#S2.1."/>
    <pre>
      The second ForCES interoperability test meeting was held by the
      IETF ForCES Working Group on February 24-25, 2011...
    </pre>
    <div class="subsection" property="dcterms:title" content=" Testbed Configuration"
         typeof="bibo:DocumentPart" about="http://localhost:8000/res/rfc/6984#S2.2.">
      <span property="bibo:chapter" content="2.2."
            about="http://localhost:8000/res/rfc/6984#S2.2."/>
      <div class="subsubsection" property="dcterms:title" content=" Participants' Access"
           typeof="bibo:DocumentPart" about="http://localhost:8000/res/rfc/6984#S2.2.1.">
        <span content="2.2.1." about="http://localhost:8000/res/rfc/6984#S2.2.1."
              property="bibo:chapter"/>
        <pre>
          NTT and ZJSU were physically present for the testing at the
          Internet Technology Lab (ITL) at Zhejiang Gongshang
          University in China...
        </pre>
      </div>
    </div>
  </div>
</div>
```

Note in particular that every section and subsection now has a defined URI (in the `@about` attribute). This will be useful later.

3.2.2 Handling citations in text

References / citations in RFC text is often of the form `"are to be interpreted as described in [RFC2119]"` (for citations to other RFCs in whole), `"as described in Section 7.1"` (for citations to other parts of the current document) or `"Section 2.4 of [RFC2045] says"` (for citations to a specific part in another document). We can define a simple grammar for these citations using pyparsing:

```
from pyparsing import Word, CaselessLiteral, nums
section_citation = (CaselessLiteral("section") + Word(nums+".").setResultsName("Sec")).setRes
rfc_citation = ("[RFC" + Word(nums).setResultsName("RFC") + "]").setResultsName("RFCRef")
section_rfc_citation = (section_citation + "of" + rfc_citation).setResultsName("SecRFCRef")
```

The above productions have named results for different parts of the citation, ie a citation of the form "Section 2.4 of [RFC2045] says" will result in the named matches Sec = "2.4" and RFC = "2045". The CitationParser class can be used to extract these matches into a dict, which is then passed to a uri formatter function like:

```
def rfc_uriformatter(parts):
    uri = ""
    if 'RFC' in parts:
         uri += self.canonical_uri(parts['RFC'].lstrip("0"))
    if 'Sec' in parts:
         uri += "#S" + parts['Sec']
    return uri
```

And to initialize a citation parser and have it run over the entire structured text, finding citations and formatting them into URIs as we go along, just use:

```
from ferenda import CitationParser, URIFormatter
citparser = CitationParser(section_rfc_citation,
                           section_citation,
                           rfc_citation)
citparser.set_formatter(URIFormatter(("SecRFCRef", rfc_uriformatter),
                                     ("SecRef", rfc_uriformatter),
                                     ("RFCRef", rfc_uriformatter)))
citparser.parse_recursive(doc.body)
```

The result of these lines is that the following block of plain text:

```
<pre>
   The behavior recommended in Section 2.5 is in line with generic error
   treatment during the IKE_SA_INIT exchange, per Section 2.21.1 of
   [RFC5996].
</pre>
```

...transform into this hyperlinked text:

```
<pre>
   The behavior recommended in <a href="#S2.5"
   rel="dcterms:references">Section 2.5</a> is in line with generic
   error treatment during the IKE_SA_INIT exchange, per <a
   href="http://localhost:8000/res/rfc/5996#S2.21.1"
   rel="dcterms:references">Section 2.21.1 of [RFC5996]</a>.
</pre>
```

Note: The uri formatting function uses `canonical_uri()` to create the base URI for each external reference. Proper design of the URIs you'll be using is a big topic, and you should think through what URIs you want to use for your documents and their parts. Ferenda provides a default implementation to create URIs from document properties, but you might want to override this.

The parse step is probably the part of your application which you'll spend the most time developing. You can start simple (like above) and then incrementally improve the end result by processing more metadata, model the semantic document structure better, and handle in-line references in text more correctly. See also *Building structured documents*, *Parsing document structure* and *Citation parsing*.

3.3 Calling `relate()`

The purpose of the `relate()` method is to make sure that all document data and metadata is properly stored and indexed, so that it can be easily retrieved in later steps. This consists of three steps: Loading all RDF metadata into a triplestore, loading all document content into a full text index, and making note of how documents refer to each other.

Since the output of parse is well structured XHTML+RDFa documents that, on the surface level, do not differ much from docrepo to docrepo, you should not have to change anything about this step.

Note: You might want to configure whether to load everything into a fulltext index – this operation takes a lot of time, and this index is not even used if createing a static site. You do this by setting `fulltextindex` to `False`, either in ferenda.ini or on the command line:

```
./ferenda-build.py rfc relate --all --fulltextindex=False
```

3.4 Calling `makeresources()`

This method needs to run at some point before generate and the rest of the methods. Unlike the other methods described above and below, which are run for one docrepo at a time, this method is run for the project as a whole (that is why it is a function in `ferenda.manager` instead of a `DocumentRepository` method). It constructs a set of site-wide resources such as minified js and css files, and configuration for the site-wide XSLT template. It is easy to run using the command-line tool:

```
$ ./ferenda-build.py all makeresources
```

If you use the API, you need to provide a list of instances of the docrepos that you're using, and the path to where generated resources should be stored:

```
from ferenda.manager import makeresources
config = {'datadir':'mydata'}
myrepos = [RFC(**config), W3C(**config]
makeresources(myrepos,'mydata/myresources')
```

3.5 Customizing `generate()`

The purpose of the `generate()` method is to create new browser-ready HTML files from the structured XHTML+RDFa files created by `parse()`. Unlike the files created by `parse()`, these files will contain site-branded headers, footers, navigation menus and such. They will also contain related content not directly found in the parsed files themselves: Sectioned documents will have a automatically-generated table of contents, and other documents that refer to a particular document will be listed in a sidebar in that document. If the references are made to individual sections, there will be sidebars for all such referenced sections.

The default implementation does this in two steps. In the first, `prep_annotation_file()` fetches metadata about other documents that relates to the document to be generated into an *annotation file*. In the second, `Transformer`

runs an XSLT transformation on the source file (which sources the annotation file and a configuration file created by `makeresources()`) in order to create the browser-ready HTML file.

You should not need to override the general `generate()` method, but you might want to control how the annotation file and the XSLT transformation is done.

3.5.1 Getting annotations

The `prep_annotation_file()` step is driven by a SPARQL construct query. The default query fetches metadata about every other document that refers to the document (or sections thereof) you're generating, using the `dcterms:references` predicate. By setting the class variable `sparql_annotations` to the file name of SPARQL query file of your choice, you can override this query.

Since our metadata contains more specialized statements on how document refer to each other, in the form of `rfc:updates` and `rfc:obsoletes` statements, we want a query that'll fetch this metadata as well. When we query for metadata about a particular document, we want to know if there is any other document that updates or obsoletes this document. Using a CONSTRUCT query, we create `rfc:isUpdatedBy` and `rfc:isObsoletedBy` references to such documents.

```
    sparql_annotations = "rfc-annotations.rq"
```

The contents of `rfc-annotations.rq`, placed in the current directory, should be:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX bibo: <http://purl.org/ontology/bibo/>
PREFIX rfc: <http://example.org/ontology/rfc/>

CONSTRUCT {?s ?p ?o .
           <%(uri)s> rfc:isObsoletedBy ?obsoleter .
           <%(uri)s> rfc:isUpdatedBy ?updater .
           <%(uri)s> dcterms:isReferencedBy ?referencer .
          }
WHERE
{
   # get all literal metadata where the document is the subject
   { ?s ?p ?o .
     # FILTER(strstarts(str(?s), "%(uri)s"))
     FILTER(?s = <%(uri)s> && !isUri(?o))
   }
   UNION
   # get all metadata (except unrelated dcterms:references) about
   #  resources that dcterms:references the document or any of its
   #  sub-resources.
   { ?s dcterms:references+ <%(uri)s> ;
        ?p ?o .
     BIND(?s as ?referencer)
     FILTER(?p != dcterms:references || strstarts(str(?o), "%(uri)s"))
   }
   UNION
   # get all metadata (except dcterms:references) about any resource that
   # rfc:updates or rfc:obsolets the document
   { ?s ?x <%(uri)s> ;
        ?p ?o .
     FILTER(?x in (rfc:updates, rfc:obsoletes) && ?p != dcterms:references)
   }
   # finally, bind obsoleting and updating resources to new variables for
   # use in the CONSTRUCT clause
```

```
   UNION { ?obsoleter rfc:obsoletes <%(uri)s> . }
   UNION { ?updater   rfc:updates   <%(uri)s> . }
}
```

Note that `%(uri)s` will be replaced with the URI for the document we're querying about.

Now, when querying the triplestore for metadata about RFC 6021, the (abbreviated) result is:

```
<graph xmlns:dcterms="http://purl.org/dc/terms/"
       xmlns:rfc="http://example.org/ontology/rfc/"
       xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <resource uri="http://localhost:8000/res/rfc/6021">
    <rfc:isObsoletedBy ref="http://localhost:8000/res/rfc/6991"/>
    <dcterms:published fmt="datatype">
      <date xmlns="http://www.w3.org/2001/XMLSchema#">2010-10-01</date>
    </dcterms:published>
    <dcterms:title xml:lang="en">Common YANG Data Types</dcterms:title>
  </resource>
  <resource uri="http://localhost:8000/res/rfc/6991">
    <a><rfc:RFC/></a>
    <rfc:obsoletes ref="http://localhost:8000/res/rfc/6021"/>
    <dcterms:published fmt="datatype">
      <date xmlns="http://www.w3.org/2001/XMLSchema#">2013-07-01</date>
    </dcterms:published>
    <dcterms:title xml:lang="en">Common YANG Data Types</dcterms:title>
  </resource>
</graph>
```

Note: You can find this file in `data/rfc/annotations/6021.grit.xml`. It's in the Grit format for easy inclusion in XSLT processing.

Even if you're not familiar with the format, or with RDF in general, you can see that it contains information about two resources: first the document we've queried about (RFC 6021), then the document that obsoletes the same document (RFC 6991).

Note: If you're coming from a relational database/SQL background, it can be a little difficult to come to grips with graph databases and SPARQL. The book "Learning SPARQL" by Bob DuCharme is highly recommended.

3.5.2 Transforming to HTML

The `Transformer` step is driven by a XSLT stylesheet. The default stylesheet uses a site-wide configuration file (created by `makeresources()`) for things like site name and top-level navigation, and lists the document content, section by section, alongside of other documents that contains references (in the form of `dcterms:references`) for each section. The SPARQL query and the XSLT stylesheet often goes hand in hand – if your stylesheet needs a certain piece of data, the query must be adjusted to fetch it. By setting he class variable `xslt_template` in the same way as you did for the SPARQL query, you can override the default.

```
    xslt_template = "rfc.xsl"
```

The contents of `rfc.xsl`, placed in the current directory, should be:

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0"
                xmlns:xhtml="http://www.w3.org/1999/xhtml"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
```

```
                xmlns:dcterms="http://purl.org/dc/terms/"
                xmlns:rfc="http://example.org/ontology/rfc/"
                xml:space="preserve"
                exclude-result-prefixes="xhtml rdf">

  <xsl:include href="base.xsl"/>

  <!-- Implementations of templates called by base.xsl -->
  <xsl:template name="headtitle"><xsl:value-of select="//xhtml:title"/> | <xsl:value-of select="$con
  <xsl:template name="metarobots"/>
  <xsl:template name="linkalternate"/>
  <xsl:template name="headmetadata"/>
  <xsl:template name="bodyclass">rfc</xsl:template>
  <xsl:template name="pagetitle">
    <h1><xsl:value-of select="../xhtml:head/xhtml:title"/></h1>
  </xsl:template>

  <xsl:template match="xhtml:a"><a href="{@href}"><xsl:value-of select="."/></a></xsl:template>

  <xsl:template match="xhtml:pre[1]">
    <pre><xsl:apply-templates/>
    </pre>
    <xsl:if test="count(ancestor::*) = 2">
      <xsl:call-template name="aside-annotations">
        <xsl:with-param name="uri" select="../@about"/>
      </xsl:call-template>
    </xsl:if>
  </xsl:template>

  <!-- everything that has an @about attribute, i.e. _is_ something
       (with a URI) gets a <section> with an <aside> for inbound links etc -->
  <xsl:template match="xhtml:div[@about]">

    <div class="section-wrapper" about="{@about}"><!-- needed? -->
      <section id="{substring-after(@about,'#')}">
        <xsl:variable name="sectionheading"><xsl:if test="xhtml:span[@property='bibo:chapter']/@conte
        <xsl:if test="count(ancestor::*) = 2">
            <h2><xsl:value-of select="$sectionheading"/></h2>
        </xsl:if>
        <xsl:if test="count(ancestor::*) = 3">
          <h3><xsl:value-of select="$sectionheading"/></h3>
        </xsl:if>
        <xsl:if test="count(ancestor::*) = 4">
          <h4><xsl:value-of select="$sectionheading"/></h4>
        </xsl:if>
        <xsl:apply-templates select="*[not(@about)]"/>
      </section>
      <xsl:call-template name="aside-annotations">
        <xsl:with-param name="uri" select="@about"/>
      </xsl:call-template>
    </div>
    <xsl:apply-templates select="xhtml:div[@about]"/>
  </xsl:template>

  <!-- remove spans which only purpose is to contain RDFa data -->
  <xsl:template match="xhtml:span[@property and @content and not(text())]"/>

  <!-- construct the side navigation -->
```

```
  <xsl:template match="xhtml:div[@about]" mode="toc">
    <li><a href="#{substring-after(@about,'#')}"><xsl:if test="xhtml:span/@content"><xsl:value-of sel
    <ul><xsl:apply-templates mode="toc"/></ul>
    </xsl:if></li>
  </xsl:template>

  <!-- named template called from other templates which match
       xhtml:div[@about] and pre[1] above, and which creates -->
  <xsl:template name="aside-annotations">
    <xsl:param name="uri"/>
    <xsl:if test="$annotations/resource[@uri=$uri]/dcterms:isReferencedBy">
      <aside class="annotations">
        <h2>References to <xsl:value-of select="$annotations/resource[@uri=$uri]/dcterms:identifier"/
        <xsl:for-each select="$annotations/resource[@uri=$uri]/rfc:isObsoletedBy">
          <xsl:variable name="referencing" select="@ref"/>
          Obsoleted by
          <a href="{@ref}">
            <xsl:value-of select="$annotations/resource[@uri=$referencing]/dcterms:identifier"/>
          </a><br/>
        </xsl:for-each>
        <xsl:for-each select="$annotations/resource[@uri=$uri]/rfc:isUpdatedBy">
          <xsl:variable name="referencing" select="@ref"/>
          Updated by
          <a href="{@ref}">
            <xsl:value-of select="$annotations/resource[@uri=$referencing]/dcterms:identifier"/>
          </a><br/>
        </xsl:for-each>
        <xsl:for-each select="$annotations/resource[@uri=$uri]/dcterms:isReferencedBy">
          <xsl:variable name="referencing" select="@ref"/>
          Referenced by
          <a href="{@ref}">
            <xsl:value-of select="$annotations/resource[@uri=$referencing]/dcterms:identifier"/>
          </a><br/>
        </xsl:for-each>
      </aside>
    </xsl:if>
  </xsl:template>

  <!-- default template: translate everything from whatever namespace
       it's in (usually the XHTML1.1 NS) into the default namespace
       -->
  <xsl:template match="*"><xsl:element name="{local-name(.)}"><xsl:apply-templates select="node()"/><

  <!-- default template for toc handling: do nothing -->
  <xsl:template match="@*|node()" mode="toc"/>

</xsl:stylesheet>
```

This XSLT stylesheet depends on `base.xsl` (which resides in `ferenda/res/xsl` in the source distribution of ferenda – take a look if you want to know how everything fits together). The main responsibility of this stylesheet is to format individual elements of the document body.

`base.xsl` takes care of the main chrome of the page, and it has a default implementation (that basically transforms everything from XHTML1.1 to HTML5, and removes some RDFa-only elements). It also loads and provides the annotation file in the global variable $annotations. The above XSLT stylesheet uses this to fetch information about referencing documents. In particular, when processing an older document, it lists if later documents have updated or obsoleted it (see the named template `aside-annotations`).

You might notice that this XSLT template flattens the nested structure of sections which we spent so much effort

to create in the parse step. This is to make it easier to put up the aside boxes next to each part of the document, independent of the nesting level.

Note: While both the SPARQL query and the XSLT stylesheet might look complicated (and unless you're a RDF/XSL expert, they are...), most of the time you can get a good result using the default generic query and stylesheet.

3.6 Customizing `toc()`

The purpose of the `toc()` method is to create a set of pages that acts as tables of contents for all documents in your docrepo. For large document collections there are often several different ways of creating such tables, eg. sorted by title, publication date, document status, author and similar. The pages uses the same site-branding,headers, footers, navigation menus etc used by `generate()`.

The default implementation is generic enough to handle most cases, but you'll have to override other methods which it calls, primarily `facets()` and `toc_item()`. These methods depend on the metadata you've created by your parse implementation, but in the simplest cases it's enough to specify that you want one set of pages organized by the `dcterms:title` of each document (alphabetically sorted) and another by `dcterms:issued` (numerically/calendarically sorted). The default implementation does exactly this.

In our case, we wish to create four kinds of sorting: By identifier (RFC number), by date of issue, by title and by category. These map directly to four kinds of metadata that we've stored about each and every document. By overriding `facets()` we can specify these four *facets*, aspects of documents used for grouping and sorting.

```
def facets(self):
    from ferenda import Facet
    return [Facet(self.ns['dcterms'].title),
            Facet(self.ns['dcterms'].issued),
            Facet(self.ns['dcterms'].subject),
            Facet(self.ns['dcterms'].identifier)]
```

After running toc with this change, you can see that three sets of index pages are created. By default, the `dcterms:identifier` predicate isn't used for the TOC pages, as it's often derived from the document title. Furthermore, you'll get some error messages along the lines of "Best Current Practice does not look like a valid URI", which is because the `dcterms:subject` predicate normally should have URIs as values, and we are using plain string literals.

We can fix both these problems by customizing our facet objects a little. We specify that we wish to use `dcterms:identifier` as a TOC facet, and provide a simple method to group RFCs by their identifier in groups of 100, ie one page for RFC 1-99, another for RFC 100-199, and so on. We also specify that we expect our `dcterms:subject` values to be plain strings.

```
def facets(self):
    def select_rfcnum(row, binding, resource_graph):
        # "RFC 6998" -> "6900"
        return row[binding][4:-2] + "00"
    from ferenda import Facet
    return [Facet(self.ns['dcterms'].title),
            Facet(self.ns['dcterms'].issued),
            Facet(self.ns['dcterms'].subject,
                  selector=Facet.defaultselector,
                  identificator=Facet.defaultselector,
                  key=Facet.defaultselector),
            Facet(self.ns['dcterms'].identifier,
                  use_for_toc=True,
                  selector=select_rfcnum,
                  pagetitle="RFC %(selected)s00-%(selected)s99")]
```

The above code gives some example of how `Facet` objects can be configured. However, a `Facet` object does not control how each individual document is listed on a toc page. The default formatting just lists the title of the document, linked to the document in question. For RFCs, who mainly is referenced using their RFC number rather than their title, we'd like to add the RFC number in this display. This is done by overriding `toc_item()`.

```
def toc_item(self, binding, row):
    from ferenda.elements import Link
    return [row['dcterms_identifier'] + ": ",
            Link(row['dcterms_title'],
                 uri=row['uri'])]
```

Se also *Customizing the table(s) of content* and *Grouping documents with facets*.

3.7 Customizing `news()`

The purpose of `news()`, the next to final step, is to provide a set of news feeds for your document repository.

The default implementation gives you one single news feed for all documents in your docrepo, and creates both browser-ready HTML (using the same headers, footers, navigation menus etc used by `generate()`) and Atom syndication format files.

You can specify some basic criteria similar to the way you specified the organization of your TOC pages, since you might want to split up the documents in different feeds, for example one feed for each RFC track.

```
def news_criteria(self):
    from ferenda import Describer, NewsCriteria
    from rdflib import Graph

    # function that returns a closure, which acts as a custom
    # selector function for the NewsCriteria objects.
    def selector_for(category):
        def selector(entry):
            graph = Graph()
            graph.parse(self.store.distilled_path(entry.basefile))
            desc = Describer(graph, entry.id)
            return desc.getvalue(self.ns['dcterms'].subject) == category
        return selector

    return [NewsCriteria('all','All RFCs'),
            NewsCriteria('informational', 'Informational RFCs',
                         selector=selector_for("Informational")),
            NewsCriteria('bcp', 'Best Current Practice RFCs',
                         selector=selector_for("Best Current Practice")),
            NewsCriteria('experimental', 'Experimental RFCs',
                         selector=selector_for("Experimental")),
            NewsCriteria('standards', 'Standards Track RFCs',
                         selector=selector_for("Standards Track"))]
```

When running `news`, this will create five different atom feeds (which are mirrored as HTML pages) under `data/rfc/news`: One containing all documents, and four others that contain documents in a particular category.

Note: As you can see, the resulting HTML pages are a little rough around the edges. Also, there isn't currently any way of discovering the Atom feeds or HTML pages from the main site – you need to know the URLs. This will all be fixed in due time.

The news generation does not make use the Facet objects that we've defined. This will be fixed in later releases of Ferenda.

Se also *Customizing the news feeds*.

3.8 Customizing `frontpage()`

Finally, `frontpage()` creates a front page for your entire site with content from the different docrepos. Each docrepos `frontpage_content()` method will be called, and should return a XHTML fragment with information about the repository and it's content. Below is a simple example that uses functionality we've used in other contexts to create a list of the five latest documents, as well as a total count of documents.

```
    def frontpage_content(self, primary=False):
        from rdflib import URIRef, Graph
        from itertools import islice
        items = ""
        for entry in islice(self.news_entries(),5):
            graph = Graph()
            with self.store.open_distilled(entry.basefile) as fp:
                graph.parse(data=fp.read())
            data = {'identifier': graph.value(URIRef(entry.id), self.ns['dcterms'].identifier).toPyth
                    'uri': entry.id,
                    'title': entry.title}
            items += '<li>%(identifier)s <a href="%(uri)s">%(title)s</a></li>' % data
        return ("""<h2><a href="%(uri)s">Request for comments</a></h2>
                   <p>A complete archive of RFCs in Linked Data form. Contains %(doccount)s documents
                   <p>Latest 5 documents:</p>
                   <ul>
                      %(items)s
                   </ul>""" % {'uri':self.dataset_uri(),
                               'items': items,
                               'doccount': len(list(self.store.list_basefiles_for("_postgenerate")))}
```

3.9 Next steps

When you have written code and customized downloading, parsing and all the other steps, you'll want to run all these steps for all your docrepos in a single command by using the special value `all` for docrepo, and again `all` for action:

```
./ferenda-build.py all all
```

By now, you should have a basic idea about the key concepts of ferenda. In the next section, *Key concepts*, we'll explore them further.

CHAPTER 4

Key concepts

4.1 Project

A collection of docrepos and configuration that is used to make a useful web site. The first step in creating a project is running `ferenda-setup <projectname>`.

A project is primarily defined by its configuration file at `<projectname>/ferenda.ini`, which specifies which docrepos are used, and settings for them as well as settings for the entire project.

A project is managed using the `ferenda-build.py` tool.

If using the API instead of these command line tools, there is no concept of a project except for what your code provides. Your client code is responsible for creating the docrepo classes and providing them with proper settings. These can be loaded from a `ferenda.ini`-style file (the `LayeredConfig` class makes this simple), be hard-coded, or handled in any other way you see fit.

4.2 Configuration

A ferenda docrepo object can be configured in two ways - either when creating the object, eg:

```
d = DocumentSource(datadir="mydata", loglevel="DEBUG",force=True)
```

Note: Parameters that is not provided when creating the object are defaulted from the built-in configuration values (see below)

Or it can be configured using the `LayeredConfig` class, which takes configuration data from three places:

- built-in configuration values (provided by `get_default_options()`)
- values from a configuration file (normally `ferenda.ini`", placed alongside `ferenda-build.py`)
- command-line parameters, eg `--force --datadir=mydata`

```
d = DocumentSource()
d.config = LayeredConfig(defaults=d.get_default_options(),
                         inifile="ferenda.ini",
                         commandline=sys.argv)
```

(This is what `ferenda-build.py` does behind the scenes)

Configuration values from the configuration file overrides built-in configuration values, and command line parameters override configuration file values.

By setting the `config` property, you override any parameters provided when creating the object.

These are the normal configuration options:

option	description	default
datadir	Directory for all downloaded/parsed etc files	'data'
patchdir	Directory containing patch files used by patch_if_needed	'patches'
parseforce	Whether to re-parse downloaded files, even if resulting XHTML1.1 files exist and are newer than downloaded files	False
compress	Whether to compress intermediate files. Can be either a empty string (don't compress) or 'bz2' (compress using bz2).	''
serializejson	Whether to serialize document data as a JSON document in the parse step.	False
generateforce	Whether to re-generate browser-ready HTML5 files, even if they exist and are newer than all dependencies	False
force	If True, overrides both parseforce and generateforce.	False
fsmdebug	Whether to display debugging information from FSMParser	False
refresh	Whether to re-download all files even if previously downloaded.	False
lastdownload	The datetime when this repo was last downloaded (stored in conf file)	None
downloadmax	Maximum number of documents to download (None means download all of them).	None
conditionalget	Whether to use Conditional GET (through the If-modified-since and/or If-none-match headers)	True
url	The basic URL for the created site, used as template for all managed resources in a docrepo (see `canonical_uri()`).	'http://localhost:8000/'
fulltextindex	Whether to index all text in a fulltext search engine. Note: This can take a lot of time.	True
useragent	The user-agent used with any external HTTP Requests. Please change this into something containing your contact info.	'ferenda-bot'
storetype	Any of the suppored types: 'SQLITE', 'SLEEPYCAT', 'SESAME' or 'FUSEKI'. See *Triple stores*.	'SQLITE'
storelocation	The file path or URL to the triple store, dependent on the storetype	'data/ferenda.sqlite'
storerepository	The repository/database to use within the given triple store (if applicable)	'ferenda'
indextype	Any of the supported types: 'WHOOSH' or 'ELASTICSEARCH'. See *Fulltext search engines*.	'WHOOSH'
indexlocation	The location of the fulltext index	'data/whooshindex'
republishsource	Whether the Atom files should contain links to the original, unparsed, source documents	False
combineresources	Whether to combine and minify all css and js files into a single file each	False
cssfiles	A list of all required css files	['http://fonts.googleapis.com/css?family=Raleway:200,100',

4.3 DocumentRepository

A document repository (docrepo for short) is a class that handles all aspects of a document collection: Downloading the documents (or aquiring them in some other way), parsing them into structured documents, and then re-generating HTML documents with added niceties, for example references from documents from other docrepos.

You add support for a new collection of documents by subclassing `DocumentRepository`. For more details, see *Creating your own document repositories*

4.4 Document

A `Document` is the main unit of information in Ferenda. A document is primarily represented in serialized form as a XHTML 1.1 file with embedded metadata in RDFa format, and in code by the `Document` class. The class has five properties:

- `meta` (a RDFLib `Graph`)
- `body` (a tree of building blocks, normally instances of `ferenda.elements` classes, representing the structure and content of the document)
- `lang` (an IETF language tag, eg `sv` or `en-GB`)
- `uri` (a string representing the canonical URI for this document)
- `basefile` (a short internal id)

The method `render_xhtml()` (which is called automatically, as long as your `parse` method use the `managedparsing()` decorator) renders a `Document` object into a XHTML 1.1+RDFa document.

4.5 Identifiers

Documents, and parts of documents, in ferenda have a couple of different identifiers, and it's useful to understand the difference and relation between them.

- `basefile`: The *internal id* for a document. This is is internal to the document repository and is used as the base for the filenames for the stored files . The basefile isn't totally random and is expected to have some relationship with a human-readable identifier for the document. As an example from the RFC docrepo, the basefile for RFC 1147 would simply be "1147". By the rules encoded in `DocumentStore`, this results in the downloaded file `rfc/downloads/1147.txt`, the parsed file `rfc/parsed/1147.xhtml` and the generated file `rfc/generated/1147.html`. Only documents themselves, not parts of documents, have basefile identifiers.
- `uri`: The *canonical URI* for a document **or** a part of a document (generally speaking, a *resource*). This identifier is used when storing data related to the resource in a triple store and a fulltext search engine, and is also used as the external URL for the document when republishing (see *The WSGI app* and also *Document URI*). URI:s for documents can be set by settings the `uri` property of the Document object. URIs for parts of documents are set by setting the `uri` property on any `elements` based object in the body tree. When rendering the document into XHTML, render_xhtml creates RDFa statements based on this property and the `meta` property.
- `dcterms:identifier`: The *human readable* identifier for a document or a part of a document. If the document has an established human-readable identifier, such as "RFC 1147" or "2003/98/EC" (The EU directive on the re-use of public sector information), the dcterms:identifier is used for this. Unlike `basefile` and `uri`, this identifier isn't set directly as a property on an object. Instead, you add a triple with `dcterms:identifier` as the predicate to the object's `meta` property, see *Parsing and representing document metadata* and also DCMI Terms.

4.6 DocumentEntry

Apart from information about what a document contains, there is also information about how it has been handled, such as when a document was first downloaded or updated from a remote source, the URL from where it came, and when it was made available through Ferenda. .This information is encapsulated in the `DocumentEntry` class. Such objects are created and updated by various methods in `DocumentRepository`. The objects are persisted to JSON files, stored alongside the documents themselves, and are used by the `news()` method in order to create valid Atom feeds.

4.7 File storage

During the course of processing, data about each individual document is stored in many different files in various formats. The `DocumentStore` class handles most aspects of this file handling. A configured DocumentStore object is available as the `store` property on any DocumentRepository object.

Example: If a created docrepo object `d` has the alias `foo`, and handles a document with the basefile identifier `bar`, data about the document is then stored:

- When downloaded, the original data as retrieved from the remote server, is stored as `data/foo/downloaded/bar.html`, as determined by `d.store.downloaded_path()`
- At the same time, a DocumentEntry object is serialized as `data/foo/entries/bar.json`, as determined by `d.store.documententry_path()`
- If the downloaded source needs to be transformed into some intermediate format before parsing (which is the case for eg. PDF or Word documents), the intermediate data is stored as `data/foo/intermediate/bar.xml`, as determined by `d.store.intermediate_path()`
- When the downloaded data has been parsed, the parsed XHTML+RDFa document is stored as `data/foo/parsed/bar.xhtml`, as determined by `d.store.parsed_path()`
- From the parsed document is automatically destilled a RDF/XML file containing all RDFa statements from the parsed file, which is stored as `data/foo/distilled/bar.rdf`, as determined by `d.store.`
  `data/foo/annotations/bar.grit.txt`, as determined by `d.store.annotation_path()`.
- During the `relate` step, all documents which are referred to by any other document are marked as dependencies of that document. If the `bar` document is dependent on another document, then this dependency is recorded in a dependency file stored at `data/foo/deps/bar.txt`, as determined by `d.store.dependencies_path()`.
- Just prior to the generation of browser-ready HTML5 files, all metadata in the system as a whole which is relevant to `bar` is serialized in an annotation file in GRIT/XML format at `data/foo/annotations/bar.grit.txt`, as determined by `d.store.annotation_path()`.
- Finally, the generated HTML5 file is created at `data/foo/generated/bar.html`, as determined by `d.store.generated_path()`. (This step also updates the serialized DocumentEntry object described above)

4.7.1 Archiving

Whenever a new version of an existing document is downloaded, an archiving process takes place when `archive()` is called (by `download_if_needed()`). This method requires a version id, which can be any string that uniquely identifies a certain revision of the document. When called, all of the above files are moved into the subdirectory in the following way (assuming that the version id is "42"):

The result of this process is that a version id for the previously existing files is calculated (by default, this is just a simple incrementing integer, but the document in your docrepo might have a more suitable version identifier already,

in which case you should override `get_archive_version()` to return this), and then all the above files (if they have been generated) are moved into the subdirectory `archive` in the following way.

`data/foo/downloaded/bar.html` -> `data/foo/archive/downloaded/bar/42.html`

The method `get_archive_version()` is used to calculate the version id. The default implementation just provides a simple incrementing integer, but if the documents in your docrepo has a more suitable version identifier already, you should override `get_archive_version()` to return this.

The archive path is calculated by providing the optional `version` parameter to any of the `*_path` methods above.

To list all archived versions for a given basefile, use the `list_versions()` method.

4.7.2 The `open_*` methods

In many cases, you don't really need to know the filename that the `*_path` methods return, because you only want to read from or write to it. For these cases, you can use the `open_*` methods instead. These work as context managers just as the builtin open method do, and can be used in the same way:

Instead of:

```
        path = self.store.downloaded_path(basefile)
        with open(path, mode="wb") as fp:
            fp.write(b"...")
```

use:

```
        with self.store.open_downloaded(path, mode="wb") as fp:
            fp.write(b"...")
```

4.7.3 Attachments

In many cases, a single file cannot represent the entirety of a document. For example, a downloaded HTML file may need a series of inline images. These can be handled as attachments by the download method. Just use the optional attachment parameter to the appropriate *_path / open_* methods:

```
from __future__ import unicode_literals
# begin part-1
class TestDocrepo(DocumentRepository):

    storage_policy = "dir"

    def download_single(self, basefile):
        mainurl = self.document_url_template % {'basefile': basefile}
        self.download_if_needed(basefile, mainurl)
        with self.store.open_downloaded(basefile) as fp:
            soup = BeautifulSoup(fp.read())
        for img in soup.find_all("img"):
            imgurl = urljoin(mainurl, img["src"])
# begin part-2
            # open eg. data/foo/downloaded/bar/hello.jpg for writing
            with self.store.open_downloaded(basefile,
                                            attachment=img["src"],
                                            mode="wb") as fp:
```

Note: The DocumentStore object must be configured to handle attachments by setting the `storage_policy`

property to `dir`. This alters the behaviour of all `*_path` methods, so that eg. the main downloaded path becomes `data/foo/downloaded/bar/index.html` instead of `data/foo/downloaded/bar.html`

To list all attachments for a document, use `list_attachments()` method.

Note that only some of the `*_path` / `open_*` methods supports the `attachment` parameter (it doesn't make sense to have attachments for DocumentEntry files or distilled RDF/XML files).

CHAPTER 5

Parsing and representing document metadata

Every document has a number of properties, such as it's title, authors, publication date, type and much more. These properties are called metadata. Ferenda does not have a fixed set of which metadata properties are available for any particular document type. Instead, it encourages you to describe the document using RDF and any suitable vocabulary (or vocabularies). If you are new to RDF, a good starting point is the RDF Primer document.

Each document has a `meta` property which initially is an empty RDFLib `Graph` object. As part of the `parse()` method, you should fill this graph with *triples* (metadata statements) about the document.

5.1 Document URI

In order to create these metadata statements, you should first create a suitable URI for your document. Preferably, this should be a URI based on the URL where your web site will be published, ie if you plan on publishing it on `http://mynetstandards.org/`, a URI for RFC 4711 might be `http://mynetstandards.org/res/rfc/4711` (ie based on the base URL, the docrepo alias, and the basefile). By changing the `url` variable in your project configuration file, you can set the base URL from which all document URIs are derived. If you wish to have more control over the exact way URIs are constructed, you can override `canonical_uri()`.

Note: In some cases, there will be another *canonical URI* for the document you're describing, used by other people in other contexts. In these cases, you should specifiy that the metadata you're publishing is about the exact same object by adding a triple of the type `owl:sameAs` with that other canonical URI as value.

The URI for any document is available as a `uri` property.

5.2 Adding metadata using the RDFLib API

With this, you can create metadata for your document using the RDFLib Graph API.

```
    # Simpler way
    def parse_metadata_from_soup(self, soup, doc):
        from ferenda import Describer
        from datetime import datetime
        title = "My Document title"
        authors = ["Fred Bloggs", "Joe Shmoe"]
        identifier = "Docno 2013:4711"
        pubdate = datetime(2013,1,6,10,8,0)
        d = Describer(doc.meta, doc.uri)
```

```
        d.rdftype(self.rdf_type)
        d.value(self.ns['prov'].wasGeneratedBy, self.qualified_class_name())
        d.value(self.ns['dcterms'].title, title, lang=doc.lang)
        d.value(self.ns['dcterms'].identifier, identifier)
        for author in authors:
            d.value(self.ns['dcterms'].author, author)
```

5.3 A simpler way of adding metadata

The default RDFLib graph API is somewhat cumbersome for adding triples to a metadata graph. Ferenda has a convenience wrapper, `Describer` (itself a subclass of `rdflib.extras.describer.Describer`) that makes this somewhat easier. The `ns` class property also contains a number of references to popular vocabularies. The above can be made more succint like this:

```
    # Simpler way
    def parse_metadata_from_soup(self, soup, doc):
        from ferenda import Describer
        from datetime import datetime
        title = "My Document title"
        authors = ["Fred Bloggs", "Joe Shmoe"]
        identifier = "Docno 2013:4711"
        pubdate = datetime(2013,1,6,10,8,0)
        d = Describer(doc.meta, doc.uri)
        d.rdftype(self.rdf_type)
        d.value(self.ns['prov'].wasGeneratedBy, self.qualified_class_name())
        d.value(self.ns['dcterms'].title, title, lang=doc.lang)
        d.value(self.ns['dcterms'].identifier, identifier)
        for author in authors:
            d.value(self.ns['dcterms'].author, author)
```

Note: `parse_metadata_from_soup()` doesn't return anything. It only modifies the `doc` object passed to it.

5.4 Vocabularies

Each RDF vocabulary is defined by a URI, and all terms (types and properties) of that vocabulary is typically directly derived from it. The vocabulary URI therefore acts as a namespace. Like namespaces in XML, a shorter prefix is often assigned to the namespace so that one can use `rdf:type` rather than `http://www.w3.org/1999/02/22-rdf-syntax-ns#type`. The DocumentRepository object keeps a dictionary of common (prefix,namespace)s in the class property `ns` – your code can modify this list in order to add vocabulary terms relevant for your documents.

5.5 Serialization of metadata

The `render_xhtml()` method serializes all information in `doc.body` and `doc.meta` to a XHTML+RDFa file (the exact location given by `parsed_path()`). The metadata specified by doc.meta ends up in the `<head>` section of this XHTML file.

The actual RDF statements are also *distilled* to a separate RDF/XML file found alongside this file (the location given by `distilled_path()`) for convenience.

5.6 Metadata about parts of the document

Just like the main Document object, individual parts of the document (represented as `ferenda.elements` objects) can have `uri` and `meta` properties. Unlike the main Document objects, these properties are not initialized beforehand. But if you do create these properties, they are used to serialize metadata into RDFa properties for each

```
    def parse_document_from_soup(self, soup, doc):
        from ferenda.elements import Page
        from ferenda import Describer
        part = Page(["This is a part of a document"],
                    ordinal=42,
                    uri="http://example.org/doc#42",
                    meta=self.make_graph())
        d = Describer(part.meta, part.uri)
        d.rdftype(self.ns['bibo'].DocumentPart)
        # the dcterms:identifier for a document part is often whatever
        # would be the preferred way to cite that part in another
        # document
        d.value(self.ns['dcterms'].identifier, "Doc:4711, p 42")
```

This results in the following document fragment:

```
<div xmlns="http://www.w3.org/1999/xhtml"
     about="http://example.org/doc#42"
     typeof="bibo:DocumentPart"
     class="page">
  <span property="dcterms:identifier"
        content="Doc:4711, p 42"
        xml:lang=""/>
     This is a part of a document
</div>
```

CHAPTER 6

Building structured documents

Any structured documents can be viewed as a tree of higher-level elements (such as chapters or sections) that contains smaller elements (like subsections or lists) that each in turn contains even smaller elements (like paragraphs or list items). When using ferenda, you can create documents by creating such trees of elements. The `ferenda.elements` module contains classes for such elements.

Most of the classes can be used like python lists (and are, in fact, subclasses of `list`). Unlike the aproach used by `xml.etree.ElementTree` and `BeautifulSoup`, where all objects are of a specific class, and a object property determines the type of element, the element objects are of different classes if the elements are different. This means that elements representing a paragraph are `ferenda.elements.Paragraph`, and elements representing a document section are `ferenda.elements.Section` and so on. The core `ferenda.elements` module contains around 15 classes that covers many basic document elements, and the submodule `ferenda.elements.html` contains classes that correspond to all HTML tags. There is some functional overlap between these two module, but `ferenda.elements` contains several constructs which aren't directly expressible as HTML elements (eg. `Page`, :~py:class:*ferenda.elements.SectionalElement* and :~py:class:*ferenda.elements.Footnote*)

Each element constructor (or at least those derived from `CompoundElement`) takes a list as an argument (same as `list`), but also any number of keyword arguments. This enables you to construct a simple document like this:

```
from ferenda.elements import Body, Heading, Paragraph, Footnote

doc = Body([Heading(["About Doc 43/2012 and it's interpretation"],predicate="dcterms:title"),
            Paragraph(["According to Doc 43/2012",
                       Footnote(["Available at http://example.org/xyz"]),
                       " the bizbaz should be frobnicated"])
           ])
```

Note: Since `CompoundElement` works like `list`, which is initialized with any iterable, you should normalliy initialize it with a single-element list of strings. If you initialize it directly with a string, the constructor will treat that string as an iterable and create one child element for every character in the string.

6.1 Creating your own element classes

The exact structure of documents differ greatly. A general document format such as XHTML or ODF cannot contain special constructs for preamble recitals of EC directives or patent claims of US patents. But your own code can create new classes for this. Example:

```
from ferenda.elements import CompoundElement, OrdinalElement

class Preamble(CompoundElement): pass
```

```
class PreambleRecital(CompoundElement,OrdinalElement):
    tagname = "div"
    rdftype = "eurlex:PreambleRecital"

doc = Preamble([PreambleRecital("Un",ordinal=1)],
               [PreambleRecital("Deux",ordinal=2)],
               [PreambleRecital("Trois",ordinal=3)])
```

6.2 Mixin classes

As the above example shows, it's possible and even recommended to use multiple inheritance to compose objects by subclassing two classes – one main class who's semantics you're extending, and one mixin class that contains particular properties. The following classes are useful as mixins:

- `OrdinalElement`: for representing elements with some sort of ordinal numbering. An ordinal element has an `ordinal` property, and different ordinal objects can be compared or sorted. The sort is based on the ordinal property. The ordinal property is a string, but comparisons/sorts are done in a natural way, i.e. "2" < "2 a" < "10".
- `TemporalElement`: for representing things that has a start and/or a end date. A temporal element has an `in_effect` method which takes a date (or uses today's date if none given) and returns true if that date falls between the start and end date.

6.3 Rendering to XHTML

The built-in classes are rendered as XHTML by the built-in method `render_xhtml()`, which first creates a `<head>` section containing all document-level metadata (i.e. the data you have specified in your documents `meta` property), and then calls the `as_xhtml` method on the root body element. The method is called with `doc.uri` as a single argument, which is then used as the RDF subject for all triples in the document (except for those sub-elements which themselves have a `uri` property)

All built-in element classes derive from `AbstractElement`, which contains a generic implementation of `as_xhtml()`, that recursively creates a lxml element tree from itself and it's children.

Your own classes can specify how they are to be rendered in XHTML by overriding the `tagname` and `classname` properties, or for full control, the `as_xhtml()` method.

As an example, the class `SectionalElement` overrides `as_xhtml` to the effect that if you provide `identifier`, `ordinal` and `title` properties for the object, a resource URI is automatically constructed and four RDF triples are created (rdf:type, dcterms:title, dcterms:identifier, and bibo:chapter):

```
from ferenda.elements import SectionalElement
p = SectionalElement(["Some content"],
                     ordinal = "1a",
                     identifier = "Doc pt 1(a)",
                     title="Title or name of the part")
body = Body([p])
from lxml import etree
etree.tostring(body.as_xhtml("http://example.org/doc"))
```

...which results in:

```
<body xmlns="http://www.w3.org/1999/xhtml" about="http://example.org/doc">
  <div about="http://example.org/doc#S1a"
```

```
       typeof="bibo:DocumentPart"
       property="dcterms:title"
       content="Title or name of the part"
       class="sectionalelement">
    <span about="http://example.org/doc#S1a"
          property="dcterms:identifier"
          content="Doc pt 1(a)" />
    <span about="http://example.org/doc#S1a"
          property="bibo:chapter"
          content="1a" />
    Some content
  </div>
</body>
```

However, this is a convenience method of SectionalElement, amd may not be appropriate for your needs. The general way of attaching metdata to document parts, as specified in *Metadata about parts of the document*, is to provide each document part with a `uri` and `meta` property. These are then automatically serialized as RDFa statements by the default `as_xhtml` implementation.

6.4 Convenience methods

Your element tree structure can be serialized to well-formed XML using the `serialize()` method. Such a serialization can be turned back into the same tree using `deserialize()`. This is primarily useful during debugging.

You might also find the `as_plaintext` method useful. It works similar to `as_xhtml`, but returns a plaintext string with the contents of an element, including all sub-elements

The `ferenda.elements.html` module contains the method `elements_from_soup()` which converts a BeautifulSoup tree into the equivalent tree of element objects.

CHAPTER 7

Parsing document structure

In many scenarios, the basic steps in parsing source documents are similar. If your source does not contain properly nested structures that accurately represent the structure of the document (such as well-authored XML documents), you will have to re-create the structure that the author intended. Usually, text formatting, section numbering and other clues contain just enough information to do that.

In many cases, your source document will naturally be split up in a large number of "chunks". These chunks may be lines or paragraphs in a plaintext documents, or tags of a certain type in a certain location in a HTML document. Regardless, it is often easy to generate a list of such chunks.

Note: For those with a background in computer science and formal languages, a chunk is sort of the same thing as a token, but whereas a token typically is a few characters in length, a chunk is typically one to several sentences long. Splitting up a documents in chunks is also typically much simpler than the process of tokenization.

These chunks can be fed to a *finite state machine*, which looks at each chunk, determines what kind of *structural element* it probably is (eg. a headline, the start of a chapter, a item in a bulleted list...) by looking at the chunk in the context of previous chunks, and then explicitly re-creates the document structure that the author (presumably) intended.

7.1 FSMParser

The framework contains a class for creating such state machines, `FSMParser`. It is used with a set of the following objects:

Object	Purpose
Recognizers	Functions that look at a chunk and determines if it is a particular structural element.
Constructors	Functions that creates a document element from a chunk (or series of chunks)
States	Identifiers for the current state of the document being parsed, ie. "in-preamble", "in-ordered-list"
Transitions	mapping (current state(s), recognizer) -> (new state, constructor)

You initialize the parser with the transition table (which contains the other objects), then call it's parse() method with a iterator of chunks, an initial state, and an initial constructor. The result of parse is a nested document object tree.

7.2 A simple example

Consider a very simple document format that only has three kinds of structural elements: a normal paragraph, preformatted text, and sections. Each section has a title and may contain paragraphs or preformatted text, which in turn may not contain anything else. All chunks are separated by double newlines

The section is identified by a header, which is any single-line string followed by a line of = characters of the same length. Any time a new header is encountered, this signals the end of the current section:

```
This is a header
================
```

A preformatted section is any chunk where each line starts with at least two spaces:

```
# some example of preformatted text
def world(name):
    return "Hello", name
```

A paragraph is anything else:

```
This is a simple paragraph.
It can contain short lines and longer lines.
```

(You might recognize this format as a very simple form of ReStructuredText).

Recognizers for these three elements are easy to build:

```
from ferenda import elements, FSMParser

def is_section(parser):
    chunk = parser.reader.peek()
    lines = chunk.split("\n")
    return (len(lines) == 2 and
            len(lines[0]) == len(lines[1]) and
            lines[1] == "=" * len(lines[0]))

def is_preformatted(parser):
    chunk = parser.reader.peek()
    lines=chunk.split("\n")
    not_indented = lambda x: not x.startswith("  ")
    return len(list(filter(not_indented,lines))) == 0

def is_paragraph(parser):
    return True
```

The `elements` module contains ready-built classes which we can use to build our constructors:

```
def make_body(parser):
    b = elements.Body()
    return parser.make_children(b)

def make_section(parser):
    chunk = parser.reader.next()
    title = chunk.split("\n")[0]
    s = elements.Section(title=title)
    return parser.make_children(s)
setattr(make_section,'newstate','section')

def make_paragraph(parser):
    return elements.Paragraph([parser.reader.next()])

def make_preformatted(parser):
    return elements.Preformatted([parser.reader.next()])
```

Note that any constructor which may contain sub-elements must itself call the `make_children()` method of the parser. That method takes a parent object, and then repeatedly creates child objects which it attaches to that parent

object, until a exit condition is met. Each call to create a child object may, in turn, call make_children (not so in this very simple example).

The final step in putting this together is defining the transition table, and then creating, configuring and running the parser:

```
transitions = {("body", is_section): (make_section, "section"),
               ("section", is_paragraph): (make_paragraph, None),
               ("section", is_preformatted): (make_preformatted, None),
               ("section", is_section): (False, None)}

text = """First section
=============

This is a regular paragraph. It will not be matched by is_section
(unlike the above chunk) or is_preformatted (unlike the below chunk),
but by the catch-all is_paragraph. The recognizers are run in the
order specified by FSMParser.set_transitions().

    This is a preformatted section.
        It could be used for source code,
    +-------------------+
    |   line drawings   |
    +-------------------+
        or what have                you.

Second section
==============

The above new section implicitly closed the first section which we
were in. This was made explicit by the last transition rule, which
stated that any time a section is encountered while in the "section"
state, we should not create any more children (False) but instead
return to our previous state (which in this case is "body", but for a
more complex language could be any number of states)."""

p = FSMParser()
p.set_recognizers(is_section, is_preformatted, is_paragraph)
p.set_transitions(transitions)
p.initial_constructor = make_body
p.initial_state = "body"
body = p.parse(text.split("\n\n"))
# print(elements.serialize(body))
```

The result of this parse is the following document object tree (passed through `serialize()`):

```
<Body>
  <Section title="First section">
    <Paragraph>
      <str>This is a regular paragraph. It will not be matched by is_section
(unlike the above chunk) or is_preformatted (unlike the below chunk),
but by the catch-all is_paragraph. The recognizers are run in the
order specified by FSMParser.set_transitions().</str>
    </Paragraph><Preformatted>
      <str>    This is a preformatted section.
        It could be used for source code,
    +-------------------+
    |   line drawings   |
    +-------------------+
```

```
        or what have                  you.</str>
    </Preformatted>
  </Section>
  <Section title="Second section">
    <Paragraph>
      <str>The above new section implicitly closed the first section which we
were in. This was made explicit by the last transition rule, which
stated that any time a section is encountered while in the "section"
state, we should not create any more children (False) but instead
return to our previous state (which in this case is "body", but for a
more complex language could be any number of states).</str>
    </Paragraph>
  </Section>
</Body>
```

7.3 Writing complex parsers

7.3.1 Recognizers

Recognizers are any callables that can be called with the parser object as only parameter (so no class- or instancemethods). Objects that implement `__call__` are OK, as are `lambda` functions.

One pattern to use when creating parsers is to have a method on your docrepo class which defines a number of nested functions, then creates a transition table using those functions, create the parser with that transition table, and then return the initialized parser object. Your main parse method can then call this method, break the input document into suitable chunks, then call parse on the recieved parser object.

7.3.2 Constructors

Like recognizers, constructors may be any callable, and they are called with the parser object as the only parameter.

Constructors that return elements which in themselves do not contain sub-elements are simple to write – just return the created element (see eg `make_paragraph` or `make_preformatted` above).

Constructors that are to return elements that may contain subelement must first create the element, then call parser.:meth:*ferenda.FSMParser.make_children* with that element as a single argument. `make_children` will treat that element as a list, and append any sub-elements created to that list, before returning it.

7.3.3 The parser object

The parser object is passed to every recognizer and constructor. The most common use is to read the next available chunk from it's reader property – this is an instance of a simple wrapper around the stream of chunks. The reader has two methods: `peek` and `next`, which both returns the next available chunk, but `next` also consumes the chunk in question. A recognizer typically calls `parser.reader.peek()`, a constructor typically calls `parser.reader.next()`.

The parser object also has the following properties

Property	Description
currentstate	The current state of the parser, using whatever value for state that was defined in the transition table (typically a string)
debug	boolean that indicates whether to emit debug messages (by default False)

There is also a `parser._debug()` method that emits debug messages, indicating current parser nesting level and current state, if `parser.debug` is `True`

7.3.4 The transition table

The transition table is a mapping between `(currentstate(s), successful recognizer)` and `(constructor-or-false,newstate-or-None)`

The transition table is used in the following way: All recognizers that can be applicable in the current state are tried in the specified order until one of them returns True. Using this pair of (currentstate, recognizer), the corresponding value tuple is looked up in the transition table.

`constructor-or-False`: ...

`newstate-or-None`: ...

The key in the transition table can also be a callable, which is called with (currentstate,symbol,parser?) and is expected to return a `(constructor-or-false,newstate-or-None)` tuple

7.4 Tips for debugging your parser

Two useful commands in the `Devel` module:

```
$ # sets debug, prints serialize(parser.parse(...))
$ ./ferenda-build.py devel fsmparse parser < chunks
$ # sets debug, returns name of matching function
$ ./ferenda-build.py devel fsmanalyze parser <currentstate> < chunk
```

CHAPTER 8

Citation parsing

In many cases, the text in the body of a document contains references (citations) to other documents in the same or related document collections. A good implementation of a document repository needs to find and express these references. In ferenda, references are expressed as basic hyperlinks which uses the `rel` attribute to specify the sort of relationship that the reference expresses. The process of citation parsing consists of analysing the raw text, finding references within that text, constructing sensible URIs for each reference, and formatting these as `<a href="..." rel="...">[citation]</a>` style links.

Since standards for expressing references / citations are very diverse, Ferenda requires that the docrepo programmer specifies the basic rules of how to recognize a reference, and how to put together the properties from a reference (such as year of publication, or page) into a URI.

8.1 The built-in solution

Ferenda uses the Pyparsing library in order to find and process citations. As an example, we'll specify citation patterns and URI formats for references that occurr in RFC documents. These are primarily of three different kinds (examples come from RFC 2616):

1. URL references, eg "GET http://www.w3.org/pub/WWW/TheProject.html HTTP/1.1"
2. IETF document references, eg "STD 3", "BCP 14" and "RFC 2068"
3. Internal endnote references, eg "[47]" and "[33]"

We'd like to make sure that any URL reference gets turned into a link to that same URL, that any IETF document reference gets turned into the canonical URI for that document, and that internal endote references gets turned into document-relative links, eg "#endnote-47" and "#endnote-33". (This requires that other parts of the `parse()` process has created IDs for these in `doc.body`, which we assume has been done).

Turning URL references in plain text into real links is so common that ferenda has built-in support for this. The support comes in two parts: First running a parser that detects URLs in the textual content, and secondly, for each match, running a URL formatter on the parse result.

At the end of your `parse()` method, do the following.

```
from ferenda import CitationParser
from ferenda import URIFormatter
import ferenda.citationpatterns
import ferenda.uriformats

# CitationParser is initialized with a list of pyparsing
# ParserElements (or any other object that has a scanString method
# that returns a generator of (tokens,start,end) tuples, where start
```

```
# and end are integer string indicies and tokens are dict-like
# objects)
citparser = CitationParser(ferenda.citationpatterns.url)

# URIFormatter is initialized with a list of tuples, where each
# tuple is a string (identifying a named ParseResult) and a function
# (that takes as a single argument a dict-like object and returns a
# URI string (possibly relative)
citparser.set_formatter(URIFormatter(("URLRef", ferenda.uriformats.url)))

citparser.parse_recursive(doc.body)
```

The `parse_recursive()` takes any `elements` document tree and modifies it in-place to mark up any references to proper `Link` objects.

8.2 Extending the built-in support

Building your own citation patterns and URI formats is fairly simple. First, specify your patterns in the form of a pyparsing parseExpression, and make sure that both the expression as a whole, and any individual significant properties, are named by calling `.setResultName`.

Then, create a set of formatting functions that takes the named properties from the parse expressions above and use them to create a URI.

Finally, initialize a `CitationParser` object from your parse expressions and a `URIFormatter` object that maps named parse expressions to their corresponding URI formatting function, and call `parse_recursive()`

```
from pyparsing import Word, nums

from ferenda import CitationParser
from ferenda import URIFormatter
import ferenda.citationpatterns
import ferenda.uriformats

# Create two ParserElements for IETF document references and internal
# references
rfc_citation = "RFC" + Word(nums).setResultsName("RFCRef")
bcp_citation = "BCP" + Word(nums).setResultsName("BCPRef")
std_citation = "STD" + Word(nums).setResultsName("STDRef")
ietf_doc_citation = (rfc_citation | bcp_citation | std_citation).setResultsName("IETFRef")

endnote_citation = ("[" + Word(nums).setResultsName("EndnoteID") + "]").setResultsName("EndnoteRef")

# Create a URI formatter for IETF documents (URI formatter for endnotes
# is so simple that we just use a lambda function below
def rfc_uri_formatter(parts):
    # parts is a dict-like object created from the named result parts
    # of our grammar, eg those ParserElement for which we've called
    # .setResultsName(), in this case eg. {'RFCRef':'2068'}

    # NOTE: If your document collection contains documents of this
    # type and you're republishing them, feel free to change these
    # URIs to URIs under your control,
    # eg. "http://mynetstandards.org/rfc/%(RFCRef)s/" and so on
    if 'RFCRef' in parts:
        return "http://www.ietf.org/rfc/rfc%(RFCRef)s.txt" % parts
    elif 'BCPRef' in parts:
```

```
            return "http://tools.ietf.org/rfc/bcp/bcp%(BCPRef)s.txt" % parts
        elif 'STDRef' in parts:
            return "http://rfc-editor.org/std/std%(STDRef)s.txt" % parts
        else:
            return None

# CitationParser is initialized with a list of pyparsing
# ParserElements (or any other object that has a scanString method
# that returns a generator of (tokens,start,end) tuples, where start
# and end are integer string indicies and tokens are dict-like
# objects)
citparser = CitationParser(ferenda.citationpatterns.url,
                           ietf_doc_citation,
                           endnote_citation)

# URIFormatter is initialized with a list of tuples, where each
# tuple is a string (identifying a named ParseResult) and a function
# (that takes as a single argument a dict-like object and returns a
# URI string (possibly relative)
citparser.set_formatter(URIFormatter(("url", ferenda.uriformats.url),
                                     ("IETFRef", rfc_uri_formatter),
                                     ("EndnoteRef", lambda d: "#endnote-%(EndnoteID)s" % d)))

citparser.parse_recursive(doc.body)
```

This turns this document

```
<body xmlns="http://www.w3.org/1999/xhtml" about="http://example.org/doc/">
  <h1>Main document</h1>
  <p>A naked URL: http://www.w3.org/pub/WWW/TheProject.html.</p>
  <p>Some IETF document references: See STD 3, BCP 14 and RFC 2068.</p>
  <p>An internal endnote reference: ...relevance ranking, cf. [47]</p>
  <h2>References</h2>
  <p id="endnote-47">47: Malmgren, Towards a theory of jurisprudential
    ranking</p>
</body>
```

Into this document:

```
<body xmlns="http://www.w3.org/1999/xhtml" about="http://example.org/doc/">
  <h1>Main document</h1>
  <p>
    A naked URL: <a href="http://www.w3.org/pub/WWW/TheProject.html"
                    rel="dcterms:references"
                    >http://www.w3.org/pub/WWW/TheProject.html</a>.
  </p>
  <p>
    Some IETF document references: See
    <a href="http://rfc-editor.org/std/std3.txt"
       rel="dcterms:references">STD 3</a>,
    <a href="http://tools.ietf.org/rfc/bcp/bcp14.txt"
       rel="dcterms:references">BCP 14</a> and
    <a href="http://www.ietf.org/rfc/rfc2068.txt"
       rel="dcterms:references">RFC 2068</a>.
     </p>
  <p>
    An internal endnote reference: ...relevance ranking, cf.
    <a href="#endnote-47"
       rel="dcterms:references">[47]</a>
```

```
  </p>
  <h2>References</h2>
  <p id="endnote-47">47: Malmgren, Towards a theory of jurisprudential
    ranking</p>
</body>
```

8.3 Rolling your own

For more complicated situations you can skip calling `parse_recursive()` and instead do your own processing with the optional support of `CitationParser`.

This is needed in particular for complicated `ParserElement` objects which may contain several sub-`ParserElement` which needs to be turned into individual links. As an example, the text "under Article 56 (2), Article 57 or Article 100a of the Treaty establishing the European Community" may be matched by a single top-level ParseResult (and probably must be, if "Article 56 (2)" is to actually reference article 56(2) in the Treaty), but should be turned in to three separate links.

In those cases, iterate through your `doc.body` yourself, and for each text part do something like the following:

```
from ferenda import CitationParser, URIFormatter, citationpatterns, uriformats
from ferenda.elements import Link

citparser = CitationParser()
citparser.add_grammar(citationpatterns.url)
formatter = URIFormatter(("url", uriformats.url))

res = []
text = "An example: http://example.org/. That is all."

for node in citparser.parse_string(text):
    if isinstance(node,str):
        # non-linked text, add and continue
        res.append(node)
    if isinstance(node, tuple):
        (text, match) = node
        uri = formatter.format(match)
        if uri:
            res.append(Link(uri, text, rel="dcterms:references"))
```

CHAPTER 9

Grouping documents with facets

A collection of documents can typically be arranged in a set of groups, such as by year of publication, by document author, or by keyword. In ferenda, each such method of grouping is described in the form of a `Facet`. By providing a list of Facet objects in its `facets()` method, your docrepo can specify multiple ways of arranging the documents it's handling. These facets are used to construct a static Table of contents for your site, as well as defining the fields available for querying when using the REST API.

A facet object is initialized with a set of parameters that together define the method of grouping. These include the RDF predicate that contains the data used for grouping, the datatype to be used for that data, functions (or other callables) that sorts the data into discrete groups, and other parameters that affect eg. the sorting order or if a particular facet is used in a particular context.

The grouping is primarily done through a selector function. The selector function recieves a dict with some basic information about one document, the name of the current facet (binding), and optionally some repo-dependent extra data in the form of an RDF graph. It should return a single string. The selector is called once (at least) for every document in the docrepo, and each resulting group contains those documents that the selector returned identical strings for. As a simple example, a selector may group documents into years of publication by finding the date of the `dcterms:issued` property and extracting the year part of it.

9.1 Contexts where facets are used

9.1.1 Table of contents

Each docrepo will have their own set of Table of contents pages. The TOC for a docrepo will contain one set of pages for each defined facet, unless `use_for_toc` is set to `False`.

9.1.2 The ReST API

The ReST API uses all defined facets for all repos simultaneously. This means that you can query eg. all documents published in a certain year, and get results from all docrepos. This requires that the defined facets don't clash, eg. that you don't have two facets based on `dcterms:publisher` where one uses URI references and the other uses.

9.1.3 The fulltext index

The metadata that each facet uses is stored as a separate field in the fulltext index. Facet can specify exactly how a particular facet should be stored (ie if the field should be boosted in any particular way). Note that the data stored in the fulltext index is not passed through the selector function, the original RDF data is stored as-is.

9.2 Grouping a document in several groups

If a docrepo uses a facet that has `multiple_values` set to `True`, it's possible for that facet to categorize the document in more than one group (a typical usecase is documents that have multiple `dcterms:subject` keywords, or articles that have multiple `dcterms:creator` authors).

9.3 Combining facets from different docrepos

Facets that map to the same fulltextindex field must be equal. The rules for equality: If the `rdftype` and the `dimension_type` and `dimension_label` and `selector` is equal, then the facets are equal. `selector` functions are only equal if they are the same function object, ie it's not just enough that they are two functions that work identically.

CHAPTER 10

Customizing the table(s) of content

In order to make the processed documents in a docrepo accessible for a website visitors, some sort of index or table of contents (TOC) that lists all available documents must be created. It's often helpful to create different lists depending on different facets of the information in documents, eg. to sort document by title, publication date, document status, author and similar properties.

Ferenda contains a number of methods that help with this task. The general process has three steps:

1. Determine the criteria for how to group and sort all documents
2. Query the triplestore for basic information about all documents in the docrepo
3. Apply these criteria on the basic information from the database

It should be noted that you don't need to do anything in order to get a very basic TOC. As long as your parse() step has extracted a `dcterms:title` string and optionally a `dcterms:issued` date for each document, you'll get basic "Sorted by title" and "Sorted by date of publication" TOCs for free. But if you've been a

10.1 Defining facets for grouping and sorting

A facet in this case is a method for grouping a set into documents into distinct categories, then sorting the documents, as well as the categories themseves.

Each facet is represented by a `Facet` object. If you want to customize the table of contents, you have to provide a list of these by overriding `facets()`.

The basic way to do this is to initialize each Facet object with a rdf predicate. Ferenda has some basic knowledge about some common predicates and know how to construct sensible Facet objects for them – ie. if you specify the predicate `dcterms:issued`, you get a Facet object that groups documents by year of publication and sorts each group by date of publication.

```
    def facets(self):
        from ferenda import Facet
        return [Facet(self.ns['dcterms'].issued),
                Facet(self.ns['dcterms'].identifier)]
```

You can customize the behaviour of each Facet by providing extra arguments to

The `label` and `pagetitle` parameters are useful to control the headings and labels for the generated pages. They should hopefully be self-explainatory.

The `selector` and `key` parameters should be functions (or any other callable) that accept a dictionary of string values, one string which is generally a key on the dictionary, and one rdflib graph containing whatever `commondata`. These functions are called once each for each row in the result set generated in the next step (see below) with the

contents of that row. They should each return a single string value. The `selector` function should return the label of a group that the document belongs to, i.e. the initial letter of the title, or the year of a publication date. The `key` function should return a value that will be used for sorting, i.e. for document titles it could return the title without any leading "The", lowercased, spaces removed etc. See also *Grouping documents with facets*.

10.2 Getting information about all documents

The next step is to perform a single SELECT query against the triplestore that retrieves a single large table, where each document is a row with a number of properties.

(This is different from the case of getting information related to a particular document, in that case, a CONSTRUCT query that retrieves a small RDF graph is used).

Your list of Facet objects returned by `facets()` is used to automatically select all data from the SPARQL store.

10.3 Making the TOC pages

The final step is to apply these criteria to the table of document properties in order to create a set of static HTML5 pages. This is in turn done in three different sub-steps, neither of which you'll have to override.

The first sub-step, `toc_pagesets()`, applies the defined criteria to the data fetched from the triple store to calculate the complete set of TOC pages needed for each criteria (in the form of a `TocPageset` object, filled with `TocPage` objects). If your criteria groups documents by year of publication date, this method will yield one page for every year that at least one document was published in.

The next sub-step, `toc_select_for_pages()`, applies the criteria on the data again, and adds each document to the appropriate `TocPage` object.

The final sub-step transforms each of these `TocPage` objects into a HTML5 file. In the process, the method `toc_item()` is called for every single document listed on every single TOC page. This method controls how each document is presented when laid out. It's called with a dict and a binding (same as used on the `selector` and `key` functions), and is expected to return a list of `elements` objects.

As an example, if you want to group by dcterms:identifier, but present each document with dcterms:identifier + dcterms:title:

```
    def toc_item(self, binding, row):
        # note: look at binding to determine which pageset is being
        # constructed in case you want to present documents in
        # different ways depending on that.
        from ferenda.elements import Link
        return [row['identifier'] + ": ",
                Link(row['title'],
                     uri=row['uri'])]
```

The generated TOC pages automatically get a visual representation of each calculated TocPageset in the left navigational column.

10.4 The first page

The main way in to each docrepos set of TOC pages is through the tabs in the main header. That link goes to a special copy of the first page in the first pageset. The order of criteria specified by `facets()` is therefore important.

CHAPTER 11

Customizing the news feeds

During the `news` step, all documents in a docrepo are published in one or more feeds. Each feed is made available in both Atom and HTML formats. You can control which feeds are created, and which documents are included in each feed, by overriding `ferenda.DocumentRepository.news_criteria()` to return a set of `NewsCriteria` objects, one for each feed. The process is similar to defining criteria for the TOC pages, but somewhat simpler and not directly connected to the data in the triple store.

Each `NewsCriteria` object should have a `basefile`, which is a slug-like short identifier for the feed. The default implementation creates a single feed named "main". It should also have a feed `title`, which is used when presenting the feed. The default implementation uses the title "All documents".

The interesting part of the `NewsCriteria` object is the `selector` parameter, which should be a function that gets called once for each document, with the corresponding `DocumentEntry` object. It should return either `True` or `False`, depending on whether this document should be included in this feed or not. The `DocumentEntry` object does not directly give access to the metadata for the document, but this information can be found by looking up the distilled RDF file that contains metadata about this document.

The entries in the feed are, by default, sorted descending on their `updated` property. If you need to override this, you can provide a `key` function which is called with a `DocumentEntry` object and returns a value that can be used for sorting.

CHAPTER 12

The WSGI app

All ferenda projects contains a built-in web application. This app provides navigation, document display and search.

12.1 Running the web application

During development, you can just `ferenda-build.py runserver`. This starts up a single-threaded web server in the foreground with the web application, by default accessible as `http://localhost:8000/`

You can also run the web application under any WSGI server, such as mod_wsgi, uWSGI or Gunicorn. `ferenda-setup` creates a file called `wsgi.py` alongside `ferenda-build.py` which is used to serve the ferenda web app using WSGI. This is the contents of that file:

```
from ferenda.manager import make_wsgi_app
inifile = os.path.join(os.path.dirname(__file__), "ferenda.ini")
application = make_wsgi_app(inifile=inifile)
```

12.1.1 Apache and mod_wsgi

In your httpd.conf:

```
WSGIScriptAlias / /path/to/project/wsgi.py
WSGIPythonPath /path/to/project
<Directory /path/to/project>
  <Files wsgi.py>
    Order deny,allow
    Allow from all
  </Files>
</Directory>
```

The ferenda web app consists mainly of static files. Only search and API requests are dynamically handled. By default though, all static files are served by the ferenda web app. This is simple to set up, but isn't optimal performance-wise.

12.1.2 Gunicorn

Just run `gunicorn wsgi:application`

12.2 URLs for retrieving resources

In keeping with Linked Data principles, all URIs for your documents should be retrievable. By default, all URIs for your documents start with `http://localhost:8000/res` (e.g. `http://localhost:8000/res/rfc/4711` – this is controlled by the `url` parameter in `ferenda.ini`). These URIs are retrievable when you run the built-in web server during development, as described above.

12.2.1 Document resources

For each resource, use the `Accept` header to retrieve different versions of it:

- `curl -H "Accept: text/html" http://localhost:8000/res/rfc/4711` returns `rfc/generated/4711.html`
- `curl -H "Accept: application/xhtml+xml" http://localhost:8000/res/rfc/4711` returns `rfc/parsed/4711.xhtml`
- `curl -H "Accept: application/rdf+xml" http://localhost:8000/res/rfc/4711` returns `rfc/distilled/4711.rdf`
- `curl -H "Accept: text/turtle" http://localhost:8000/res/rfc/4711` returns `rfc/distilled/4711.rdf`, but in Turtle format
- `curl -H "Accept: text/plain" http://localhost:8000/res/rfc/4711` returns `rfc/distilled/4711.rdf`, but in NTriples format

You can also get *extended information* about a single document in various RDF flavours. This extended information includes everything that `construct_annotations()` returns, i.e. information about documents that refer to this document.

- `curl -H "Accept: application/rdf+xml" http://localhost:8000/res/rfc/4711/data` returns a RDF/XML combination of `rfc/distilled/4711.rdf` and `rfc/annotation/4711.grit.xml`
- `curl -H "Accept: text/turtle" http://localhost:8000/res/rfc/4711/data` returns the same in Turtle format
- `curl -H "Accept: text/plain" http://localhost:8000/res/rfc/4711/data` returns the same in NTriples format
- `curl -H "Accept: application/json" http://localhost:8000/res/rfc/4711/data` returns the same in JSON-LD format.

12.2.2 Dataset resources

Each docrepo exposes information about the data it contains through it's dataset URI. This is a single URI (controlled by `dataset_uri()`) which can be queried in a similar way as the document resources above:

- `curl -H "Accept: application/html" http://localhost/dataset/rfc` returns a HTML view of a Table of Contents for all documents (see *Customizing the table(s) of content*)
- `curl -H "Accept: text/plain" http://localhost/dataset/rfc` returns `rfc/distilled/dump.nt` which contains all RDF statements for all documents in the repository.
- `curl -H "Accept: application/rdf+xml" http://localhost/dataset/rfc` returns the same, but in RDF/XML format.
- `curl -H "Accept: text/turtle" http://localhost/dataset/rfc` returns the same, but in turtle format.

12.2.3 File extension content negotiation

In some environments, it might be difficult to set the Accept header. Therefore, it's also possible to request different versions of a resource using a file extension suffix. Ie. requesting `http://localhost:8000/res/base/123.ttl` gives the same result as requesting the resource `http://localhost:8000/res/base/123` using the `Accept: text/turtle` header. The following extensions can be used

Content-type	Extension
application/xhtml+xml	.xhtml
application/rdf+xml	.rdf
text/turtle	.ttl
text/plain	.nt
application/json	.json

See also *The ReST API for querying*.

CHAPTER 13

The ReST API for querying

Ferenda tries to adhere to Linked Data principles, which makes it easy to explain how to get information about any individual document or any complete dataset (see *URLs for retrieving resources*). Sometimes it's desirable to query for all documents matching a particular criteria, including full text search. Ferenda has a simple API, based on the `rinfo-service` component of RDL, and inspired by Linked data API, that enables you to do that. This API only provides search/select operations that returns a result list. For information about each individual result in that list, use the methods described in *URLs for retrieving resources*.

Note: Much of the things described below are also possible to do in pure SPARQL. Ferenda does not expose any open SPARQL endpoints to the world, though. But if you find the below API lacking in some aspect, it's certainly possible to directly expose your chosen triplestores SPARQL endpoint (as long as you're using Fuseki or Sesame) to the world.

The default endpoint to query is your main URL + `/api/`, eg. `http://localhost:8000/api/`. The requests always use GET and encode their parameters in the URL, and the responses are always in JSON format.

13.1 Free text queries

The simplest form of query is a free text query that is run against all text of all documents. Use the parameter `q`, eg. `http://localhost:8000/api/?q=tail` returns all documents (and document fragments) containing the word "tail".

13.2 Result lists

The result of a query will be a JSON document containing some general properties of the result, and a list of result items, eg:

```
{
    "current": "/myapi/?q=tail",
    "duration": null,
    "items": [
        {
            "dcterms_identifier": "123(A)",
            "dcterms_issued": "2014-01-04",
            "dcterms_publisher": {
                "iri": "http://example.org/publisher/A",
                "label": "http://example.org/publisher/A"
            },
```

```
            "dcterms_title": "Example",
            "matches": {
                "text": "<em class=\"match\">tail</em> end of the main document"
            },
            "rdf_type": "http://purl.org/ontology/bibo/Standard",
            "iri": "http://example.org/base/123/a"
        }
    ],
    "itemsPerPage": 10,
    "startIndex": 0,
    "totalResults": 1
}
```


Each result item contain all fields that have been indexed (as specified by your docrepos' facets, see *Grouping documents with facets*, the document URI (as the field `iri`) and optionally a field `matches` that provides a snipped of the matching text.

13.3 Parameters

Any indexed property, as defined by your facets, can be used for querying. The parameter is the same as the qname for the rdftype with `_` instead of `:`, eq to search all documents that have `dcterms:publisher` set to `'http://example.org/publisher/A`, use `http://localhost:8000/api/?dcterms_publisher=http%3A%2F%2Fexample.org%2Fpublisher%2FA`

You can use * as a wildcard for any string data, eg. the above could be shortened to `http://localhost:8000/api/?dcterms_publisher=*%2Fpublisher%2FA`.

If you have a facet with a set `dimension_label`, you can use that label directly as a parameter, eg `http://localhost:8000/api/?aprilfools=true`.

13.4 Paging

By default, the result list only contains 10 results. You can inspect the properties `startIndex` and `totalResults` of the response to find out if there are more results, and use the special parameter `_page` to request subsequent pages of results. You can also request a different length of the result list through the `_pageSize` parameter.

13.5 Statistics

By requesting the special resource `;stats`, eg `http://localhost:8000/api/;stats`, you can get a statistics view over all documents in all your docrepos for each of your defined facets including the number of document for each value of it's selector, eg:


```
{
    "type": "DataSet",
    "slices": [
        {
            "dimension": "rdf_type",
            "observations": [
                {"count": 3,
                 "term": "bibo:Standard"}
            ]
        },
```

```
        {
            "dimension": "dcterms_publisher",
            "observations": [ {
                "count": 1,
                "ref": "http://example.org/publisher/A"
            }, {
                "count": 2,
                "ref": "http://example.org/publisher/B"
            } ]
        }, {
            "dimension": "dcterms_issued",
            "observations": [ {
                "count": 1,
                "year": "2013"
            }, {
                "count": 2,
                "year": "2014"
            } ]
        } ]
}
```

You can also get the same information for the documents in any result list by setting the special parameter `_stats=on`.

13.6 Ranges

For some parameters, particularly those that use datetime values, it's useful to specify ranges instead of exact values. By prefixing the parameter name with `min-`, `max-` or `year-`, it's possible to do that, eg. `http://localhost:8000/api/?min-dcterms_issued=2012-04-01` to retrieve all documents that have a dcterms:issued later than 2012-04-01, or `http://localhost:8000/api/?year-dcterms_issued=2012` to retrieve all documents that are dct:issued during 2012.

13.7 Support resources

The special resources `common.json` and `terms.json` (eg. `http://localhost:8000/api/common.json` and `http://localhost:8000/api/terms.json`) contains all the extra data (see *Custom common data*) and ontologies (see *Custom ontologies*) that your repositories use, in JSON-LD format. You can use these to display user-friendly labels for properties and things in your application.

13.8 Legacy mode

Ferenda can be made directly compatible with the API used by `rinfo-service` (mentioned above) by activating the setting `legacyapi`, eg by setting `legacyapi = True` in ferenda.conf or using the option `--legacyapi` on the command line.

Note that this setting is used both during the `makeresources` step as well as when serving the API eg with the `runserver` command. If you want to play with this setting, you'll need to re-run `makeresources --force` with this enabled.

Running `makeresources` with this setting enabled also installs a API explorer app, taken from `rinfo-service`. You can try it out at `http://localhost:8000/rsrc/ui/`.

CHAPTER 14

Setting up external databases

Ferenda stores data in three substantially different ways:

- Documents are stored in the file system
- RDF Metadata is stored in in a triple store
- Document text is stored in a fulltext search engine.

There are many capable and performant triple stores and fulltext search engines available, and ferenda supports a few of them. The default choice for both are embedded solutions (using RDFLib + SQLite for a triple store and Whoosh for a fulltext search engine) so that you can get a small system going without installing and configuring additional server processess. However, these choices do not work well with medium to large datasets, so when you start feeling that indexing and searching is getting slow, you should run an external triplestore and an external fulltext search engine.

If you're using the project framework, you set the configuration values `storetype` and `indextype` to new values. You'll find that the `ferenda-setup` tool creates a `ferenda.ini` that specifies `storetype` and `indextype`, based on whether it can find Fuseki, Sesame and/or ElasticSearch running on their default ports on localhost. You still might have to do extra configuration, particularly if you're using Sesame as a triple store.

If you setup any of the external databases after running `ferenda-setup`, or you want to use some other configuration than what `ferenda-setup` selected for you, you can still set the configuration values in `ferenda.ini` by editing the file as described below.

If you are running any of the external databases, but in a non-default location (including remote locations) you can set the environment variables `FERENDA_TRIPLESTORE_LOCATION` and/or `FERENDA_FULLTEXTINDEX_LOCATION` to the full URL before running `ferenda-setup`.

14.1 Triple stores

There are four choices.

14.1.1 RDFLib + SQLite

In `ferenda.ini`:

```
[__root__]
storetype = SQLITE
storelocation = data/ferenda.sqlite # single file
storerepository = <projectname>
```

This is the simplest way to get up and running, requiring no configuration or installs on any platform.

14.1.2 RDFLib + Sleepycat (aka bsddb)

In ferenda.ini:

```
[__root__]
storetype = SLEEPYCAT
storelocation = data/ferenda.db # directory
storerepository = <projectname>
```

This requires that bsddb (part of the standard library for python 2) or bsddb3 (separate package) is available and working (which can be a bit of pain on many platforms). Furthermore it's less stable and slower than RDFLib + SQLite, so it can't really be recommended. But since it's the only persistant storage directly supported by RDFLib, it's supported by Ferenda as well.

14.1.3 Sesame

In ferenda.ini:

```
[__root__]
storetype = SESAME
storelocation = http://localhost:8080/openrdf-sesame
storerepository = <projectname>
```

Sesame is a framework and a set of java web applications that normally runs within a Tomcat application server. If you're comfortable with Tomcat and servlet containers you can get started with this quickly, see their installation instructions. You'll need to install both the actual Sesame Server and the OpenRDF workbench.

After installing it and configuring ferenda.ini to use it, you'll need to use the OpenRDF workbench app (at http://localhost:8080/openrdf-workbench by default) to create a new repository. The recommended settings are:

```
Type: Native Java store
ID: <projectname> # eg same as storerepository in ferenda.ini
Title: Ferenda repository for <projectname>
Triple indexes: spoc,posc,cspo,opsc,psoc
```

It's much faster than the RDFLib-based stores and is fairly stable (although Ferenda's usage patterns seem to sometimes make simple operations take a disproportionate amount of time).

14.1.4 Fuseki

In ferenda.ini:

```
[__root__]
storetype = SESAME
storelocation = http://localhost:3030
storerepository = ds
```

Fuseki is a simple java server that implements most SPARQL standards and can be run without any complicated setup. It can keep data purely in memory or store it on disk. The above configuration works with the default configuration of Fuseki - just download it and run fuseki-server

Fuseki seems to be the fastest triple store that Ferenda supports, at least with Ferendas usage patterns. Since it's also the easiest to set up, it's the recommended triple store once RDFLib + SQLite isn't enough.

14.2 Fulltext search engines

There are two choices.

14.2.1 Whoosh

In `ferenda.ini`:

```
[__root__]
indextype = WHOOSH
indexlocation = data/whooshindex
```

Whoosh is an embedded python fulltext search engine, which requires no setup (it's automatically installed when installing ferenda with `pip` or `easy_install`), works reasonably well with small to medium amounts of data, and performs quick searches. However, once the index grows beyond a few hundred MB, indexing of new material begins to slow down.

14.2.2 Elasticsearch

In `ferenda.ini`:

```
[__root__]
indextype = ELASTICSEARCH
indexlocation = http://localhost:9200/ferenda/
```

Elasticsearch is a distributed fulltext search engine in java which can run in a distributed fashion and which is accessed through a simple JSON/REST API. It's easy to setup – just download it and run `bin/elasticsearch` as per the instructions. Ferenda's support for Elasticsearch is new and not yet stable, but it should be able to handle much larger amounts of data.

CHAPTER 15

Advanced topics

15.1 Composite docrepos

In some cases, a document collection may available from multiple sources, with varying degrees of completeness and/or quality. For example, in a collection of US patents, some patents may be available in structured XML with good metadata through a easy-to-use API, some in tag-soup style HTML with no metadata, requiring screenscraping, and some in the form of TIFF files that you scanned yourself. The implementation of both download() and parse() will differ wildly for these sources. You'll have something like this:

```
from ferenda import DocumentRepository, CompositeRepository
from ferenda.decorators import managedparsing

class XMLPatents(DocumentRepository):
    alias = "patxml"

    def download(self, basefile = None):
        download_from_api()

    @managedparsing
    def parse(self,doc):
        return self.transform_patent_xml_to_xhtml(doc)

class HTMLPatents(DocumentRepository):
    alias = "pathtml"

    def download(self, basefile=None):
        screenscrape()

    @managedparsing
    def parse(self,doc):
        return self.analyze_tagsoup(doc)

class ScannedPatents(DocumentRepository):
    alias = "patscan"

    # Assume that we, when we scanned the documents, placed them in their
    # correct place under data/patscan/downloaded

    def download(self, basefile=None): pass

    @managedparsing
    def parse(self,doc):
```

```
        x = self.ocr_and_structure(doc)
        return True
```

But since the result of all three parse() implementations are XHTML1.1+RDFa documents (possibly with varying degrees of data fidelity), the implementation of generate() will be substantially the same. Furthermore, you probably want to present a unified document collection to the end user, presenting documents derived from structured XML if they're available, documents derived from tagsoup HTML if an XML version wasn't available, and finally documents derived from your scanned documents if nothing else is available.

The class `CompositeRepository` makes this possible. You specify a number of subordinate docrepo classes using the `subrepos` class property.

```
class CompositePatents(CompositeRepository):
    alias = "pat"
    # Specify the classes in order of preference for parsed documents.
    # Only if XMLPatents does not have a specific patent will HTMLPatents
    # get the chance to provide it through it's parse method
    subrepos = XMLPatents, HTMLPatents, ScannedPatents

    def generate(self, basefile, otherrepos=[]):
        # Optional code to transform parsed XHTML1.1+RDFa documents
        # into browser-ready HTML5, regardless of wheter these are
        # derived from structured XML, tagsoup HTML or scanned
        # TIFFs. If your parse() method can make these parsed
        # documents sufficiently alike and generic, you might not need
        # to implement this method at all.
        self.do_the_work(basefile)
```

The CompositeRepository docrepo then acts as a proxy for all of your specialized repositories:

```
$ ./ferenda-build.py patents.CompositePatents enable
# calls download() for all subrepos
$ ./ferenda-build.py pat download
# selects the best subrepo that has patent 5,723,765, calls parse()
# for that, then copies the result to pat/parsed/ 5723765 (or links)
$ ./ferenda-build.py pat parse 5723765
# uses the pat/parsed/5723765 data. From here on, we're just like any
# other docrepo.
$ ./ferenda-build.py pat generate 5723765
```

Note that `patents.XMLPatents` and the other subrepos are never registered in ferenda.ini''. They're just called behind-the-scenes by `patents.CompositePatents`.

15.2 Patch files

It is not uncommon that source documents in a document repository contains formatting irregularities, sensitive information that must be redacted, or just outright errors. In some cases, your parse implementation can detect and correct these things, but in other cases, the irregularities are so uncommon or unique that this is not possible to do in a general way.

As an alternative, you can patch the source document (or it's intermediate representation) before the main part of your parsing logic.

The method `patch_if_needed()` automates most of this work for you. It expects a basefile and the corresponding source document as a string, looks in a *patch directory* for a corresponding patch file, and applies it if found.

By default, the patch directory is alongside the data directory. The patch file for document foo in repository bar should be placed in `patches/bar/foo.patch`. An optional description of the patch (as a plaintext, UTF-8 encoded file) can be placed in `patches/bar/foo.desc`.

`patch_if_needed()` returns a tuple (text, description). If there was no available patch, text is identical to the text passed in and description is None. If there was a patch available and it applied cleanly, text is the patched text and description is a description of the patch (or "(No patch description available)"). If there was a patch, but it didn't apply cleanly, a PatchError is raised.

Note: There is a `mkpatch` command in the Devel class which aims to automate the creation of patch files. It does not work at the moment.

15.3 External annotations

Ferenda contains a general docrepo class that fetches data from a separate MediaWiki server and stores this as annotations/descriptions related to the documents in your main docrepos. This makes it possible to present a source document and commentary on it (including annotations about individual sections) side-by-side.

See `ferenda.sources.general.MediaWiki`

15.4 Keyword hubs

Ferenda also contains a general docrepo class that lists all keywords used by documents in your main docrepos (by default, it looks for all `dcterms:subject` properties used in any document) and generate documents for each of them. These documents have no content of their own, but act as hub pages that list all documents that use a certain keyword in one place.

When used together with the MediaWiki module above, this makes it possible to write editorial descriptions about each keyword used, that is presented alongside the list of documents that use that keyword.

See `ferenda.sources.general.Keyword`

15.5 Custom common data

In many cases, you want to describe documents using references to other things that are not documents, but which should be named using URIs rather than plain text literals. This includes things like companies, publishing entities, print series and abstract things like the topic/keyword of a document. You can define a RDF graph containing more information about each such thing that you know of beforehand, eg if we want to model that some RFCs are published in the Internet Architecture Board (IAB) stream, we can define the following small graph:

```
<http://localhost:8000/ext/iab> a foaf:Organization;
    foaf:name "Internet Architecture Board (IAB)";
    skos:altLabel "IAB";
    foaf:homepage <https://www.iab.org/> .
```

If this is placed in `res/extra/[alias].ttl`, eg `res/extra/rfc.ttl`, the graph is made available as `commondata`, and is also provided as the third `resource_graph` argument to any selector/key functions of your `Facet` objects.

15.6 Custom ontologies

Some parts of ferenda, notably *The ReST API for querying*, can make use of ontologies that your docrepo uses. This is so far only used to provide human-readable descriptions of predicates used (as determined by `rdfs:label` or `rdfs:comment`). Ferenda will try to find an ontology for any namespace you use in `namespaces`, and directly supports many common vocabularies (`bibo`, `dc`, `dcterms`, `foaf`, `prov`, `rdf`, `rdfs`, `schema` and `skos`). If you have defined your own custom ontology, place it (in Turtle format) as `res/vocab/[alias].ttl`, eg. `res/vocab/rfc.ttl` to make Ferenda read it.

CHAPTER 16

API reference

16.1 Classes

16.1.1 The `DocumentRepository` class

class ferenda.**DocumentRepository**(***kwargs*)

Base class for downloading, parsing and generating HTML versions of a repository of documents.

Start building your application by subclassing this class, and then override methods in order to customize the downloading, parsing and generation behaviour.

Parameters ****kwargs** – Any named argument overrides any similarly-named configuration file parameter.

Example:

```
>>> class MyRepo(DocumentRepository):
...     alias="myrepo"
...
>>> d = MyRepo(datadir="/tmp/ferenda")
>>> d.store.downloaded_path("mybasefile").replace(os.sep,'/')
'/tmp/ferenda/myrepo/downloaded/mybasefile.html'
```

Note: This class has a ridiculous amount of methods that you can override to control most of Ferendas behaviour in all stages. For basic usage, you need only a fraction of them. Please don't be intimidated/horrified.

---

downloaded_suffix = u'.html'

File suffix for the main document format. Determines the suffix of downloaded files.

storage_policy = u'file'

Some repositories have documents in several formats, documents split amongst several files or embedded resources. If `storage_policy` is set to `dir`, then each document gets its own directory (the default filename being `index` +suffix), otherwise each doc gets stored as a file in a directory with other files. Affects `ferenda.DocumentStore.path()` (and therefore all other `*_path` methods)

alias = u'base'

A short name for the class, used by the command line `ferenda-build.py` tool. Also determines where to store downloaded, parsed and generated files. When you subclass `DocumentRepository` you *must* override this.

namespaces = [u'rdf', u'rdfs', u'xsd', u'xsi', u'dcterms', u'skos', u'foaf', u'xhv', u'owl', u'prov', u'bibo']

The namespaces that are included in the XHTML and RDF files generated by `parse()`. This can be a

list of strings, in which case the strings are assumed to be well-known prefixes to established namespaces, or a list of *(prefix, namespace)* tuples. All well-known prefixes are available in `ferenda.util.ns`.

If you specify a namespace for a well-known ontology/vocabulary, that onlology will be available as a `Graph` from the `ontologies` property.

`required_predicates` = [rdflib.term.URIRef(u'http://www.w3.org/1999/02/22-rdf-syntax-ns#type')]
A list of RDF predicates that should be present in the outdata. If any of these are missing from the result of `parse()`, a warning is logged. You can add to this list as a form of simple validation of your parsed data.

`start_url` = u'http://example.org/'
The main entry page for the remote web store of documents. May be a list of documents, a search form or whatever. If it's something more complicated than a simple list of documents, you need to override `download()` in order to tell which documents are to be downloaded.

`document_url_template` = u'http://example.org/docs/%(basefile)s.html'
A string template for creating URLs for individual documents on the remote web server. Directly used by `remote_url()` and indirectly by `download_single()`.

`document_url_regex` = u'http://example.org/docs/(?P<basefile>\\w+).html'
A regex that matches URLs for individual documents – the reverse of what `document_url_template` is used for. Used by `download()` to find suitable links if `basefile_regex` doesn't match. Must define the named group `basefile` using the `(?P<basefile>...)` syntax

`basefile_regex` = u'^ID: ?(?P<basefile>[\\w\\d\\:\\/]+)$'
A regex for matching document names in link text, as used by `download()`. Must define a named group `basefile`, just like `document_url_template`.

`rdf_type` = rdflib.term.URIRef(u'http://xmlns.com/foaf/0.1/Document')
The RDF type of the documents you are handling (expressed as a `rdflib.term.URIRef` object).

Note: If your repo produces documents of several different types, you can define this as a list (or other iterable) of `URIRef` objects. `faceted_data()` will only find documents that are any of the types.

`source_encoding` = u'utf-8'
The character set that the source HTML documents use (if applicable).

`lang` = u'en'
The language which the source documents are assumed to be written in (unless otherwise specified), and the language which output document should use.

`parse_content_selector` = u'body'
CSS selector used to select the main part of the document content by the default `parse()` implementation.

`parse_filter_selectors` = [u'script']
CSS selectors used to filter/remove certain parts of the document content by the default `parse()` implementation.

`xslt_template` = u'res/xsl/generic.xsl'
A template used by `generate()` to transform the XML file into browser-ready HTML. If your document type is complex, you might want to override this (and write your own XSLT transform). You should include `base.xslt` in that template, though.

`sparql_annotations` = u'res/sparql/annotations.rq'
A template SPARQL CONSTRUCT query for document annotations.

`documentstore_class`
alias of `DocumentStore`

ontologies
Provides a Graph loaded with the ontologies/vocabularies that this docrepo uses (as determined by the namespaces ` property).

If you're using your own vocabularies, you can place them (in Turtle format) as res/vocab/[prefix].ttl to have them loaded into the graph.

Note: Some system-like vocabularies (rdf, rdfs and owl) are never loaded into the graph.

commondata
Provides a Graph containing any extra data that is common to documents in this docrepo – this can be information about different entities that publishes the documents, the printed series in which they're published, and so on. The data is taken from res/extra/[repoalias].ttl.

config
The LayeredConfig object that contains the current configuration for this docrepo instance. You can read or write individual properties of this object, or replace it with a new LayeredConfig object entirely.

lookup_resource(*label*, *predicate=rdflib.term.URIRef(u'http://xmlns.com/foaf/0.1/name')*, *cutoff=0.8*, *warn=True*)
Given a textual identifier (ie. the name for something), lookup the canonical uri for that thing in the RDF graph containing extra data (i.e. the graph that commondata provides). The graph should have a *foaf:name`'* statement about the url with the sought label as the object.

Since data is imperfect, the textual label may be spelled or expressed different in different contexts. This method therefore performs fuzzy matching (using difflib.get_close_matches()) using the cutoff parameter determines exactly how fuzzy this matching is.

If no resource matches the given label, a KeyError is raised.

Parameters

- **label** (*str*) – The textual label to lookup
- **predicate** (*rdflib.term.RIRef*) – The RDF predicate to use when looking for the label
- **cutoff** (*float*) – How fuzzy the matching may be (1 = must match exactly, 0 = anything goes)
- **warn** (*bool*) – Whether to log a warning when an inexact match is performed

Returns The matching resource

Return type rdflib.URIRef

get_default_options()
Returns the class' configuration default configuration properties. These can be overridden by a configution file, or by named arguments to __init__(). See *Configuration* for a list of standard configuration properties (your subclass is free to define and use additional configuration properties).

Returns default configuration properties

Return type dict

classmethod **setup**(*action*, *config*)
Runs before any of the *_all methods starts executing. It just calls the appropriate setup method, ie if *action* is parse, then this method calls parse_all_setup (if defined) with the *config* object as single parameter.

classmethod **teardown**(*action*, *config*)
Runs after any of the *_all methods has finished executing. It just calls the appropriate teardown method,

ie if *action* is `parse`, then this method calls `parse_all_teardown` (if defined) with the *config* object as single parameter.

get_archive_version(*basefile*)
Get a version identifier for the current version of the document identified by `basefile`.

The default implementation simply increments most recent archived version identifier, starting at "1". If versions in your docrepo are normally identified in some other way (such as SCM revision numbers, dates or similar) you should override this method to return those identifiers.

Parameters **basefile** (*str*) – The basefile of the document to archive

Returns The version identifier for the current version of the document.

Return type str

qualified_class_name()
The qualified class name of this class

Returns class name (e.g. `ferenda.DocumentRepository`)

Return type str

canonical_uri(*basefile*)
The canonical URI for the document identified by `basefile`.

Returns The canonical URI

Return type str

dataset_uri(*param=None*, *value=None*)
Returns the URI that identifies the dataset that this docrepository provides. The default implementation is based on the url config parameter and the alias attribute of the class, c.f. `http://localhost:8000/dataset/base`.

Parameters

- **param** – An optional parameter name represeting a way of createing a subset of the dataset (eg. all document whose title starts with a particular letter)
- **value** – A value for *param* (eg. "a")

```
>>> d = DocumentRepository()
>>> d.alias
'base'
>>> d.config.url = "http://example.org/"
>>> d.dataset_uri()
'http://example.org/dataset/base'
>>> d.dataset_uri("title","a")
'http://example.org/dataset/base?title=a'
```

basefile_from_uri(*uri*)
The reverse of `canonical_uri()`. Returns `None` if the uri doesn't map to a basefile in this repo.

```
>>> d = DocumentRepository()
>>> d.alias
'base'
>>> d.config.url = "http://example.org/"
>>> d.basefile_from_uri("http://example.org/res/base/123/a")
'123/a'
>>> d.basefile_from_uri("http://example.org/res/base/123/a#S1")
'123/a'
>>> d.basefile_from_uri("http://example.org/res/other/123/a") # None
```

dataset_params_from_uri(*uri*)

Given a parametrized dataset URI, return the parameter and value used (or an empty tuple, if it is a dataset URI handled by this repo, but without any parameters).

```
>>> d = DocumentRepository()
>>> d.alias
'base'
>>> d.config.url = "http://example.org/"
>>> d.dataset_params_from_uri("http://example.org/dataset/base?title=a")
('title', 'a')
>>> d.dataset_params_from_uri("http://example.org/dataset/base")
()
```

download(**args*, ***kwargs*)

Downloads all documents from a remote web service.

The default generic implementation assumes that all documents are linked from a single page (which has the url of `start_url`), that they all have URLs matching the `document_url_regex` or that the link text is always equal to basefile (as determined by `basefile_regex`). If these assumptions don't hold, you need to override this method.

If you do override it, your download method should read and set the `lastdownload` parameter to either the datetime of the last download or any other module-specific string (id number or similar).

You should also read the `refresh` parameter. If it is `True` (the default), then you should call `download_single()` for every basefile you encounter, even though they may already exist in some form on disk. `download_single()` will normally be using conditional GET to see if there is a newer version available.

See *Writing your own download implementation* for more details.

Returns True if any document was downloaded, False otherwise.

Return type bool

download_get_basefiles(*params*)

Given *source* (a iterator that provides (element, attribute, link, pos) tuples, like `lxml.etree.iterlinks()`), generate tuples (basefile, link) for all document links found in *source*.

download_single(*basefile*, *url=None*)

Downloads the document from the web (unless explicitly specified, the URL to download is determined by `document_url_template` combined with basefile, the location on disk is determined by the function `downloaded_path()`).

If the document exists on disk, but the version on the web is unchanged (determined using a conditional GET), the file on disk is left unchanged (i.e. the timestamp is not modified).

Parameters

- **basefile** (*string*) – The basefile of the document to download
- **url** (*str*) – The URL to download (optional)

Returns `True` if the document was downloaded and stored on disk, `False` if the file on disk was not updated.

download_if_needed(*url*, *basefile*, *archive=True*, *filename=None*, *sleep=1*)

Downloads a remote resource to a local file. If a different version is already in place, archive that old version.

Parameters

- **url** (*str*) – The url to download
- **basefile** (*str*) – The basefile of the document to download
- **archive** (*bool*) – Whether to archive existing older versions of the document, or just delete the previously downloaded file.
- **filename** (*str*) – The filename to download to. If not provided, the filename is derived from the supplied basefile

Returns True if the local file was updated (and archived), False otherwise.

Return type bool

download_is_different(*existing*, *new*)
Returns True if the new file is semantically different from the existing file.

remote_url(*basefile*)
Get the URL of the source document at it's remote location, unless the source document is fetched by other means or if it cannot be computed from basefile only. The default implementation uses `document_url_template` to calculate the url.

Example:

```
>>> d = DocumentRepository()
>>> d.remote_url("123/a")
'http://example.org/docs/123/a.html'
>>> d.document_url_template = "http://mysite.org/archive/%(basefile)s/"
>>> d.remote_url("123/a")
'http://mysite.org/archive/123/a/'
```

Parameters **basefile** (*str*) – The basefile of the source document

Returns The remote url where the document can be fetched, or `None`.

Return type str

generic_url(*basefile*, *maindir*, *suffix*)
Analogous to `ferenda.DocumentStore.path()`, calculate the full local url for the given basefile and stage of processing.

Parameters

- **basefile** (*str*) – The basefile for which to calculate the local url
- **maindir** (*str*) – The processing stage directory (normally `downloaded`, `parsed`, or `generated`)
- **suffix** (*str*) – The file extension including period (i.e. `.txt`, not `txt`)

Returns The local url

Return type str

downloaded_url(*basefile*)
Get the full local url for the downloaded file for the given basefile.

Parameters **basefile** (*str*) – The basefile for which to calculate the local url

Returns The local url

Return type str

```
>>> d = DocumentRepository()
>>> d.downloaded_url("123/a")
'http://localhost:8000/base/downloaded/123/a.html'
```

classmethod parse_all_setup(*config*)
Runs any action needed prior to parsing all documents in a docrepo. The default implementation does nothing.

Note: This is a classmethod for now (and that's why a config object is passsed as an argument), but might change to a instance method.

---

classmethod parse_all_teardown(*config*)
Runs any cleanup action needed after parsing all documents in a docrepo. The default implementation does nothing.

Note: Like `parse_all_setup()` this might change to a instance method.

---

parseneeded(*basefile*)
Returns True iff there is a need to parse the given basefile. If the resulting parsed file exists and is newer than the downloaded file, there is typically no reason to parse the file.

parse(*basefile*)
Parse downloaded documents into structured XML and RDF.

It will also save the same RDF statements in a separate RDF/XML file.

You will need to provide your own parsing logic, but often it's easier to just override parse_from_soup (assuming your indata is in a HTML format parseable by BeautifulSoup) and let the base class read and write the files.

If your data is not in a HTML format, or BeautifulSoup is not an appropriate parser to use, override this method.

Parameters **doc** (*ferenda.Document*) – The document object to fill in.

soup_from_basefile(*basefile*, *encoding=u'utf-8'*, *parser=u'lxml'*)
Load the downloaded document for basefile into a BeautifulSoup object

Parameters

- **basefile** (*str*) – The basefile for the downloaded document to parse
- **encoding** (*str*) – The encoding of the downloaded document

Returns The parsed document as a `BeautifulSoup` object

Note: Helper function. You probably don't need to override it.

---

parse_metadata_from_soup(*soup*, *doc*)
Given a BeautifulSoup document, retrieve all document-level metadata from it and put it into the given `doc` object's `meta` property.

Note: The default implementation sets `rdf:type`, `dcterms:title`, `dcterms:identifier` and `prov:wasGeneratedBy` properties in `doc.meta`, as well as setting the language of the document in `doc.lang`.

Parameters

- **soup** – A parsed document, as `BeautifulSoup` object
- **doc** (*ferenda.Document*) – Our document

Returns None

parse_document_from_soup(*soup*, *doc*)
Given a BeautifulSoup document, convert it into the provided `doc` object's `body` property as suitable `ferenda.elements` objects.

Note: The default implementation respects `parse_content_selector` and `parse_filter_selectors`.

Parameters

- **soup** – A parsed document as a `BeautifulSoup` object
- **doc** (*ferenda.Document*) – Our document

Returns None

patch_if_needed(*basefile*, *text*)
Given *basefile* and the entire *text* of the downloaded or intermediate document, find if there exists a patch file under `self.config.patchdir`, and if so, applies it. Returns (patchedtext, patchdescription) if so, (text,None) otherwise.

make_document(*basefile=None*)
Create a `Document` objects with basic initialized fields.

Note: Helper method used by the `makedocument()` decorator.

Parameters **basefile** (*str*) – The basefile for the document

Return type ferenda.Document

make_graph()
Initialize a rdflib Graph object with proper namespace prefix bindings (as determined by `namespaces`)

Return type rdflib.Graph

create_external_resources(*doc*)
Optionally create external files that go together with the parsed file (stylesheets, images, etc).

The default implementation does nothing.

Parameters **doc** (*ferenda.Document*) – The document

render_xhtml(*doc*, *outfile=None*)
Renders the parsed object structure as a XHTML file with RDFa attributes (also returns the same XHTML as a string).

Parameters

- **doc** (*ferenda.Document*) – The document to render
- **outfile** (*str*) – The file name for the XHTML document

Returns The XHTML document

Return type str

render_xhtml_tree(*doc*)
Renders the parsed object structure as a `lxml.etree._Element` object.

Parameters **doc** (*ferenda.Document*) – The document to render

Returns The XHTML document as a lxml structure

Return type lxml.etree._Element

parsed_url(*basefile*)
Get the full local url for the parsed file for the given basefile.

Parameters **basefile** (*str*) – The basefile for which to calculate the local url

Returns The local url

Return type str

distilled_url(*basefile*)
Get the full local url for the distilled RDF/XML file for the given basefile.

Parameters **basefile** (*str*) – The basefile for which to calculate the local url

Returns The local url

Return type str

classmethod **relate_all_setup**(*config*)
Runs any cleanup action needed prior to relating all documents in a docrepo. The default implementation clears the corresponsing context (see `dataset_uri()`) in the triple store.

Note: Like `parse_all_setup()` this might change to a instance method.

Returns False if no relation needs to be done (as determined by the timestamp on the dump nt file)

classmethod **relate_all_teardown**(*config*)
Runs any cleanup action needed after relating all documents in a docrepo. The default implementation dumps all RDF data loaded into the triplestore into one giant N-Triples file.

Note: Like `parse_all_setup()` this might change to a instance method.

---

relate(*basefile*, *otherrepos*=[ ])
Runs various indexing operations for the document represented by *basefile*: insert RDF statements into a triple store, add this document to the dependency list to all documents that it refers to, and put the text of the document into a fulltext index.

relate_triples(*basefile*)
Insert the (previously distilled) RDF statements into the triple store.

Parameters **basefile** (*str*) – The basefile for the document containing the RDF statements.

Returns None

relate_dependencies(*basefile*, *repos*=[ ])
For each document that the basefile document refers to, attempt to find this document in the current or any other docrepo, and add the parsed document path to that documents dependency file.

add_dependency(*basefile*, *dependencyfile*)
Add the *dependencyfile* to *basefile* s dependency file. Returns True if anything new was added, False otherwise

relate_fulltext(*basefile*, *repos=None*)
Index the text of the document into fulltext index. Also indexes all metadata that facets() indicate should be indexed.

Parameters **basefile** (*str*) – The basefile for the document to be indexed.

Returns None

facets()
Provides a list of `Facet` objects that specify how documents in your docrepo should be grouped.

Override this if you want to specify your own way of grouping data in your docrepo.

faceted_data()
Provides a list of dicts, each containing a row of information about a single document in the repository. The exact fields provided are controlled by the list of `Facet` objects returned by `facet()`.

Note: The same document can occur multiple times if any of it's facets have `multiple_values` set, once for each different values that that facet has.

---

facet_query(*context*)
Constructs a SPARQL SELECT query that fetches all information needed to create faceted data.

Parameters **context** (*str*) – The context (named graph) to which to limit the query.

Returns The SPARQL query

Return type str

Example:

```
>>> d = DocumentRepository()
>>> expected = """PREFIX dcterms: <http://purl.org/dc/terms/>
... PREFIX foaf: <http://xmlns.com/foaf/0.1/>
... PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
...
... SELECT DISTINCT ?uri ?rdf_type ?dcterms_title ?dcterms_publisher ?dcterms_identifier ?dc
... FROM <http://example.org/ctx/base>
... WHERE {
...     ?uri rdf:type foaf:Document .
...     OPTIONAL { ?uri rdf:type ?rdf_type . }
...     OPTIONAL { ?uri dcterms:title ?dcterms_title . }
...     OPTIONAL { ?uri dcterms:publisher ?dcterms_publisher . }
...     OPTIONAL { ?uri dcterms:identifier ?dcterms_identifier . }
...     OPTIONAL { ?uri dcterms:issued ?dcterms_issued . }
...
... }"""
>>> d.facet_query("http://example.org/ctx/base") == expected
True
```

facet_select(*query*)
Select all data from the triple store needed to create faceted data.

Parameters **context** (*str*) – The context (named graph) to restrict the query to. If None, search entire triplestore.

Returns The results of the query, as python objects

Return type set of dicts

classmethod generate_all_setup(*config*)
Runs any action needed prior to generating all documents in a docrepo. The default implementation does

nothing.

Note: Like `parse_all_setup()` this might change to a instance method.

classmethod **generate_all_teardown**(*config*)
Runs any cleanup action needed after generating all documents in a docrepo. The default implementation does nothing.

Note: Like `parse_all_setup()` this might change to a instance method.

generate(*basefile*, *otherrepos*=[])
Generate a browser-ready HTML file from structured XML and RDF.

Uses the XML and RDF files constructed by `ferenda.DocumentRepository.parse()`.

The generation is done by XSLT, and normally you won't need to override this, but you might want to provide your own xslt file and set `ferenda.DocumentRepository.xslt_template` to the name of that file.

If you want to generate your browser-ready HTML by any other means than XSLT, you should override this method.

Parameters **basefile** (*str*) – The basefile for which to generate HTML

Returns None

get_url_transform_func(*repos*, *basedir*)

prep_annotation_file(*basefile*)
Helper function used by `generate()` – prepares a RDF/XML file containing statements that in some way annotates the information found in the document that generate handles, like URI/title of other documents that refers to this one.

Parameters **basefile** (*str*) – The basefile for which to collect annotating statements.

Returns The full path to the prepared RDF/XML file

Return type str

construct_annotations(*uri*)
Construct a RDF graph containing metadata by running the query provided by `construct_sparql_query()`

construct_sparql_query(*uri*)
Construct a SPARQL query that will select metadata relating to *uri* in some way, using the query template specified by `sparql_annotations`

graph_to_annotation_file(*graph*)
Converts a RDFLib graph into a XML file with the same statements, ordered using the Grit format (https://code.google.com/p/oort/wiki/Grit) for easier XSLT inclusion.

Parameters **graph** (*rdflib.graph.Graph*) – The graph to convert

Returns A serialized XML document with the RDF statements

Return type str

annotation_file_to_graph(*annotation_file*)
Converts a annotation file (using the Grit format) back into an RDFLib graph.

Parameters **graph** (*str*) – The filename of a serialized XML document with RDF statements

Returns The RDF statements as a regular graph

Return type rdflib.Graph

generated_url(*basefile*)
Get the full local url for the generated file for the given basefile.

Parameters **basefile** (*str*) – The basefile for which to calculate the local url

Returns The local url

Return type str

toc(*otherrepos*=[])
Creates a set of pages that together acts as a table of contents for all documents in the repository. For smaller repositories a single page might be enough, but for repositoriees with a few hundred documents or more, there will usually be one page for all documents starting with A, starting with B, and so on. There might be different ways of browseing/drilling down, i.e. both by title, publication year, keyword and so on.

The default implementation calls `faceted_data()` to get all data from the triple store, `facets()` to find out the facets for ordering, `toc_pagesets()` to calculate the total set of TOC html files, `toc_select_for_pages()` to create a list of documents for each TOC html file, and finally `toc_generate_pages()` to create the HTML files. The default implemention assumes that documents have a title (in the form of a `dcterms:title` property) and a publication date (in the form of a `dcterms:issued` property).

You can override any of these methods to customize any part of the toc generation process. Often overriding `facets()` to specify other document properties will be sufficient.

toc_pagesets(*data*, *facets*)
Calculate the set of needed TOC pages based on the result rows

Parameters

- **data** – list of dicts, each dict containing metadata about a single document
- **facets** – list of Facet objects

Returns The link text, page title and base file for each needed TOC page, structured by selection facets.

Return type 3-dimensional named tuple

Example:

```
>>> d = DocumentRepository()
>>> from rdflib.namespace import DCTERMS
>>> rows = [{'uri':'http://ex.org/1','dcterms_title':'Abc','dcterms_issued':'2009-04-02'},
...         {'uri':'http://ex.org/2','dcterms_title':'Abcd','dcterms_issued':'2010-06-30'},
...         {'uri':'http://ex.org/3','dcterms_title':'Dfg','dcterms_issued':'2010-08-01'}]
>>> from rdflib.namespace import DCTERMS
>>> facets = [Facet(DCTERMS.title), Facet(DCTERMS.issued)]
>>> pagesets=d.toc_pagesets(rows,facets)
>>> pagesets[0].label
'Sorted by title'
>>> pagesets[0].pages[0]
<TocPage binding=dcterms_title linktext=a title=Documents starting with "a" value=a>
>>> pagesets[0].pages[0].linktext
'a'
>>> pagesets[0].pages[0].title
'Documents starting with "a"'
>>> pagesets[0].pages[0].binding
'dcterms_title'
>>> pagesets[0].pages[0].value
```

```
'a'
>>> pagesets[1].label
'Sorted by publication year'
>>> pagesets[1].pages[0]
<TocPage binding=dcterms_issued linktext=2009 title=Documents published in 2009 value=2009>
```

toc_select_for_pages(*data*, *pagesets*, *facets*)

Go through all data rows (each row representing a document) and, for each toc page, select those documents that are to appear in a particular page.

Example:

```
>>> d = DocumentRepository()
>>> rows = [{'uri':'http://ex.org/1','dcterms_title':'Abc','dcterms_issued':'2009-04-02'},
...         {'uri':'http://ex.org/2','dcterms_title':'Abcd','dcterms_issued':'2010-06-30'},
...         {'uri':'http://ex.org/3','dcterms_title':'Dfg','dcterms_issued':'2010-08-01'}]
>>> from rdflib.namespace import DCTERMS
>>> facets = [Facet(DCTERMS.title), Facet(DCTERMS.issued)]
>>> pagesets=d.toc_pagesets(rows,facets)
>>> expected={('dcterms_title','a'):[[Link('Abc',uri='http://ex.org/1')],
...                                  [Link('Abcd',uri='http://ex.org/2')]],
...           ('dcterms_title','d'):[[Link('Dfg',uri='http://ex.org/3')]],
...           ('dcterms_issued','2009'):[[Link('Abc',uri='http://ex.org/1')]],
...           ('dcterms_issued','2010'):[[Link('Abcd',uri='http://ex.org/2')],
...                                      [Link('Dfg',uri='http://ex.org/3')]]}
>>> d.toc_select_for_pages(rows, pagesets, facets) == expected
True
```

Parameters

- **data** – List of dicts as returned by `toc_select()`
- **pagesets** – Result from `toc_pagesets()`
- **facets** – Result from `facets()`

Returns mapping between toc basefile and documentlist for that basefile

Return type dict

toc_item(*binding*, *row*)

Returns a formatted version of row, using Element objects

toc_generate_pages(*pagecontent*, *pagesets*, *otherrepos*=[])

Creates a set of TOC pages by calling `toc_generate_page()`.

Parameters

- **pagecontent** – Result from `toc_select_for_pages()`
- **pagesets** – Result from `toc_pagesets()`
- **otherrepos** – A list of document repository instances

toc_generate_first_page(*pagecontent*, *pagesets*, *otherrepos*=[])

Generate the main page of TOC pages.

toc_generate_page(*binding*, *value*, *documentlist*, *pagesets*, *effective_basefile=None*, *otherrepos*=[])

Generate a single TOC page.

Parameters

- **binding** – The binding used (eg. 'title' or 'issued')
- **value** – The value for the used binding (eg. 'a' or '2013'
- **documentlist** – Result from `toc_select_for_pages()`
- **pagesets** – Result from `toc_pagesets()`
- **effective_basefile** – Place the resulting page somewhere else than `toc/*binding*/*value*.html`
- **otherrepos** – A list of document repository instances

news(*otherrepos*=[])
Create a set of Atom feeds and corresponding HTML pages for new/updated documents in different categories in the repository.

news_criteria()
Returns a list of NewsCriteria objects.

news_entries()
Return a generator of all available entries, represented as tuples of (DocumentEntry, rdflib.Graph) objects. The Graph contains all distilled metadata about the document.

news_write_atom(*entries*, *title*, *basefile*, *archivesize=1000*)
Given a list of Atom entry-like objects, including links to RDF and PDF files (if applicable), create a rinfo-compatible Atom feed, optionally splitting into archives.

frontpage_content(*primary=False*)
If the module wants to provide any particular content on the frontpage, it can do so by returning a XHTML fragment (in text form) here. If primary is true, the caller wants the module to take primary responsibility for the frontpage content. If primary is false, the caller only expects a smaller amount of content (like a smaller presentation of the repository and the document it contains).

status(*basefile=None*, *samplesize=3*)
Prints out some basic status information about this repository.

get_status()
Returns basic data about the state about this repository, used by `status()`.

tabs()
Get the navigation menu segment(s) provided by this docrepo

Returns a list of tuples, where each tuple will be rendered as a tab in the main UI. First element of the tuple is the link text, and the second is the link destination. Normally, a module will only return a single tab.

Returns List of tuples

Example:

```
>>> d = DocumentRepository()
>>> d.tabs()
[('base', 'http://localhost:8000/dataset/base')]
```

footer()
Get a list of resources provided by this repo for publication in the site footer.

Works like `tabs()`, but normally returns an empty list. The repo `ferenda.sources.general.Static` is an exception.

http_handle(*environ*)
Used by the WSGI support to indicate if this repo can provide a response to a particular request. If so, returns a tuple *(fp, length, memtype)*, where *fp* is an open file of the document to be returned.

16.1.2 The Document class

class ferenda.**Document**(*meta=None*, *body=None*, *uri=None*, *lang=None*, *basefile=None*)
A document represents the content of a document together with a RDF graph containing metadata about the document. Don't create instances of Document directly. Create them through make_document() in order to properly initialize the meta property.

Parameters

- **meta** – A RDF graph containing metadata about the document
- **body** – A list of ferenda.elements based objects representing the content of the document
- **uri** – The canonical URI for this document
- **lang** – The main language of the document as a IETF language tag, i.e. "sv" or "en-GB"
- **basefile** – The basefile of the document

16.1.3 The DocumentEntry class

class ferenda.**DocumentEntry**(*path=None*)
This class has two primary uses – it is used to represent and store aspects of the downloading of each document (when it was initially downloaded, optionally updated, and last checked, as well as the URL from which it was downloaded). It's also used by the news_* methods to encapsulate various aspects of a document entry in an atom feed. Some properties and methods are used by both of these use cases, but not all.

Parameters **path** (*str*) – If this file path is an existing JSON file, the object is initialized from that file.

orig_created = None
The first time we fetched the document from it's original location.

id = None
The canonical uri for the document.

basefile = None
The basefile for the document.

orig_updated = None
The last time the content at the original location of the document was changed.

orig_checked = None
The last time we accessed the original location of this document, regardless of wheter this led to an update.

orig_url = None
The main url from where we fetched this document.

published = None
The date our parsed/processed version of the document was published.

updated = None
The last time our parsed/processed version changed in any way (due to the original content being updated, or due to changes in our parsing functionality.

title = None
A title/label for the document, as used in an Atom feed.

summary = None
A summary of the document, as used in an Atom feed.

url = None
The URL to the browser-ready version of the page, equivalent to what `generated_url()` returns.

content = None
A dict that represents metadata about the document file.

link = None
A dict that represents metadata about the document RDF metadata (such as it's URI, length, MIME-type and MD5 hash).

save (*path=None*)
Saves the state of the documententry to a JSON file at *path*. If *path* is not provided, uses the path that the object was initialized with.

set_content (*filename*, *url*, *mimetype=None*, *inline=False*)
Sets the `content` property and calculates md5 hash for the file

Parameters

- **filename** – The full path to the document file
- **url** – The full external URL that will be used to get the same document file
- **mimetype** – The MIME-type used in the atom feed. If not provided, guess from file extension.
- **inline** – whether to inline the document content in the file or refer to *url*

set_link (*filename*, *url*, *mimetype=None*)
Sets the `link` property and calculate md5 hash for the RDF metadata.

Parameters

- **filename** – The full path to the RDF file for a document
- **url** – The full external URL that will be used to get the same RDF file
- **mimetype** – The MIME-type used in the atom feed. If not provided, guess from file extension.

calculate_md5 (*filename*)
Given a filename, return the md5 value for the file's content.

guess_type (*filename*)
Given a filename, return a MIME-type based on the file extension.

16.1.4 The `DocumentStore` class

class `ferenda.`**DocumentStore** (*datadir*, *downloaded_suffix=u'.html'*, *storage_policy=u'file'*)
Unifies handling of reading and writing of various data files during the `download`, `parse` and `generate` stages.

Parameters

- **datadir** (*str*) – The root directory (including docrepo path segment) where files are stored.
- **downloaded_suffix** (*str*) – File suffix for the main source document format. Determines the suffix of downloaded files.

- **storage_policy** (*str*) – Some repositories have documents in several formats, documents split amongst several files or embedded resources. If `storage_policy` is set to `dir`, then each document gets its own directory (the default filename being `index` +suffix), otherwise each doc gets stored as a file in a directory with other files. Affects `path()` (and therefore all other `*_path` methods)

path(*basefile*, *maindir*, *suffix*, *version=None*, *attachment=None*, *storage_policy=None*)

Calculate a full filesystem path for the given parameters.

Parameters

- **basefile** (*str*) – The basefile of the resource we're calculating a filename for
- **maindir** (*str*) – The stage of processing, e.g. `downloaded` or `parsed`
- **suffix** – Appropriate file suffix, e.g. `.txt` or `.pdf`
- **version** (*str*) – Optional. The archived version id
- **attachment** (*str*) – Optional. Any associated file needed by the main file.
- **storage_policy** – Optional. Used to override *storage_policy* if needed

---

Note: This is a generic method with many parameters. In order to keep your code tidy and and loosely coupled to the actual storage policy, you should use methods like `downloaded_path()` or `parsed_path()` when possible.

Example:

```
>>> d = DocumentStore(datadir="/tmp/base")
>>> realsep = os.sep
>>> os.sep = "/"
>>> d.path('123/a', 'parsed', '.xhtml') == '/tmp/base/parsed/123/a.xhtml'
True
>>> d.storage_policy = "dir"
>>> d.path('123/a', 'parsed', '.xhtml') == '/tmp/base/parsed/123/a/index.xhtml'
True
>>> d.path('123/a', 'downloaded', None, 'r4711', 'appendix.txt') == '/tmp/base/archive/downl
True
>>> os.sep = realsep
```

Parameters

- **basefile** (*str*) – The basefile for which to calculate the path
- **maindir** – The processing stage directory (normally `downloaded`, `parsed`, or `generated`)
- **suffix** (*str*) – The file extension including period (i.e. `.txt`, not `txt`)
- **version** (*str*) – Optional, the archived version id
- **attachment** (*str*) – Optional. Any associated file needed by the main file. Requires that `storage_policy` is set to `dir`. `suffix` is ignored if this parameter is used.

Returns The full filesystem path

Return type str

open(**args*, ***kwds*)

Context manager that opens files for reading or writing. The parameters are the same as for `path()`, and the note is applicable here as well – use `open_downloaded()`, `open_parsed()` et al if possible.

Example:

```
>>> store = DocumentStore(datadir="/tmp/base")
>>> with store.open('123/a', 'parsed', '.xhtml', mode="w") as fp:
...     res = fp.write("hello world")
>>> os.path.exists("/tmp/base/parsed/123/a.xhtml")
True
```

list_basefiles_for(*action*, *basedir=None*)
Get all available basefiles that can be used for the specified action.

Parameters

- **action** (*str*) – The action for which to get available basefiles (`parse`, `relate`, `generate` or `news`)
- **basedir** (*str*) – The base directory in which to search for available files. If not provided, defaults to `self.datadir`.

Returns All available basefiles

Return type generator

list_versions(*basefile*, *action=None*)
Get all archived versions of a given basefile.

Parameters

- **basefile** (*str*) – The basefile to list archived versions for
- **action** (*str*) – The type of file to look for (either `downloaded`, `parsed` or `generated`. If `None`, look for all types.

Returns All available versions for that basefile

Return type generator

list_attachments(*basefile*, *action*, *version=None*)
Get all attachments for a basefile in a specified state

Parameters

- **action** (*str*) – The state (type of file) to look for (either `downloaded`, `parsed` or `generated`. If `None`, look for all types.
- **basefile** (*str*) – The basefile to list attachments for
- **version** (*str*) – The version of the basefile to list attachments for. If None, list attachments for the current version.

Returns All available attachments for the basefile

Return type generator

basefile_to_pathfrag(*basefile*)
Given a basefile, returns a string that can safely be used as a fragment of the path for any representation of that file. The default implementation recognizes a number of characters that are unsafe to use in file names and replaces them with HTTP percent-style encoding.

Example:

```
>>> d = DocumentStore("/tmp")
>>> realsep = os.sep
>>> os.sep = "/"
>>> d.basefile_to_pathfrag('1998:204') == '1998/%3A204'
```

```
True
>>> os.sep = realsep
```

If you wish to override how document files are stored in directories, you can override this method, but you should make sure to also override `pathfrag_to_basefile()` to work as the inverse of this method.

Parameters **basefile** (*str*) – The basefile to encode

Returns The encoded path fragment

Return type str

pathfrag_to_basefile(*pathfrag*)

Does the inverse of `basefile_to_pathfrag()`, that is, converts a fragment of a file path into the corresponding basefile.

Parameters **pathfrag** (*str*) – The path fragment to decode

Returns The resulting basefile

Return type str

archive(*basefile*, *version*)

Moves the current version of a document to an archive. All files related to the document are moved (downloaded, parsed, generated files and any existing attachment files).

Parameters

- **basefile** (*str*) – The basefile of the document to archive
- **version** (*str*) – The version id to archive under

downloaded_path(*basefile*, *version=None*, *attachment=None*)

Get the full path for the downloaded file for the given basefile (and optionally archived version and/or attachment filename).

Parameters

- **basefile** (*str*) – The basefile for which to calculate the path
- **version** (*str*) – Optional. The archived version id
- **attachment** (*str*) – Optional. Any associated file needed by the main file.

Returns The full filesystem path

Return type str

open_downloaded(*basefile*, *mode='r'*, *version=None*, *attachment=None*)

Opens files for reading and writing, c.f. `open()`. The parameters are the same as for `downloaded_path()`.

documententry_path(*basefile*, *version=None*)

Get the full path for the documententry JSON file for the given basefile (and optionally archived version).

Parameters

- **basefile** (*str*) – The basefile for which to calculate the path
- **version** (*str*) – Optional. The archived version id

Returns The full filesystem path

Return type str

intermediate_path(*basefile*, *version=None*, *attachment=None*)

Get the full path for the main intermediate file for the given basefile (and optionally archived version).

Parameters

- **basefile** (*str*) – The basefile for which to calculate the path
- **version** (*str*) – Optional. The archived version id
- **attachment** – Optional. Any associated file created or retained in the intermediate step

Returns The full filesystem path

Return type str

open_intermediate(*basefile*, *mode=u'r'*, *version=None*, *attachment=None*)
Opens files for reading and writing, c.f. `open()`. The parameters are the same as for `intermediate_path()`.

parsed_path(*basefile*, *version=None*, *attachment=None*)
Get the full path for the parsed XHTML file for the given basefile.

Parameters

- **basefile** (*str*) – The basefile for which to calculate the path
- **version** (*str*) – Optional. The archived version id
- **attachment** (*str*) – Optional. Any associated file needed by the main file (created by `parse()`)

Returns The full filesystem path

Return type str

open_parsed(*basefile*, *mode=u'r'*, *version=None*, *attachment=None*)
Opens files for reading and writing, c.f. `open()`. The parameters are the same as for `parsed_path()`.

serialized_path(*basefile*, *version=None*, *attachment=None*)
Get the full path for the serialized JSON file for the given basefile.

Parameters

- **basefile** (*str*) – The basefile for which to calculate the path
- **version** (*str*) – Optional. The archived version id

Returns The full filesystem path

Return type str

open_serialized(*basefile*, *mode=u'r'*, *version=None*)
Opens files for reading and writing, c.f. `open()`. The parameters are the same as for `serialized_path()`.

distilled_path(*basefile*, *version=None*)
Get the full path for the distilled RDF/XML file for the given basefile.

Parameters

- **basefile** (*str*) – The basefile for which to calculate the path
- **version** (*str*) – Optional. The archived version id

Returns The full filesystem path

Return type str

open_distilled(*basefile*, *mode=u'r'*, *version=None*)
Opens files for reading and writing, c.f. `open()`. The parameters are the same as for `distilled_path()`.

generated_path(*basefile*, *version=None*, *attachment=None*)
Get the full path for the generated file for the given basefile (and optionally archived version and/or attachment filename).

Parameters

- **basefile** (*str*) – The basefile for which to calculate the path
- **version** (*str*) – Optional. The archived version id
- **attachment** (*str*) – Optional. Any associated file needed by the main file.

Returns The full filesystem path

Return type str

annotation_path(*basefile*, *version=None*)
Get the full path for the annotation file for the given basefile (and optionally archived version).

Parameters

- **basefile** (*str*) – The basefile for which to calculate the path
- **version** (*str*) – Optional. The archived version id

Returns The full filesystem path

Return type str

open_annotation(*basefile*, *mode=u'r'*, *version=None*)
Opens files for reading and writing, c.f. `open()`. The parameters are the same as for `annotation_path()`.

dependencies_path(*basefile*)
Get the full path for the dependency file for the given basefile

Parameters **basefile** (*str*) – The basefile for which to calculate the path

Returns The full filesystem path

Return type str

open_dependencies(*basefile*, *mode=u'r'*)
Opens files for reading and writing, c.f. `open()`. The parameters are the same as for `dependencies_path()`.

atom_path(*basefile*)
Get the full path for the atom file for the given basefile

Note: This is used by `ferenda.DocumentRepository.news()` and does not really operate on "real" basefiles. It might be removed. You probably shouldn't use it unless you override `news()`

Parameters **basefile** (*str*) – The basefile for which to calculate the path

Returns The full filesystem path

Return type str

16.1.5 The `Facet` class

class `ferenda.`**`Facet`**(*rdftype=rdflib.term.URIRef(u'http://purl.org/dc/terms/title')*, *label=None*, *pagetitle=None*, *indexingtype=None*, *selector=None*, *key=None*, *identificator=None*, *toplevel_only=None*, *use_for_toc=None*, *selector_descending=None*, *key_descending=None*, *multiple_values=None*, *dimension_type=None*, *dimension_label=None*)

Create a facet from the given rdftype and some optional parameters.

Parameters

- **rdftype** (*rdflib.term.URIRef*) – The type of facet being created
- **label** (*str*) – A template for the label property of TocPageset objects created from this facet
- **pagetitle** (*str*) – A template for the title property of TocPage objects created from this facet
- **indexingtype** (*ferenda.fulltext.IndexedType*) – Object specifying how to store the data selected by this facet in the fulltext index
- **selector** (*callable*) – A function that takes *(row, binding, resource_graph)* and returns a string acting as a category of some kind
- **key** (*callable*) – A function that takes *(row, binding, resource_graph)* and returns a string usable for sorting
- **toplevel_only** (*bool*) – Whether this facet should be applied to documents only, or any named (ie. given an URI) fragment of a document.
- **use_for_toc** (*bool*) – Whether this facet should be used for TOC generation
- **selector_descending** (*bool*) – Whether the values returned by `selector` should be presented in lexical descending order
- **key_descending** (*bool*) – Whether documents, when sorted through the `key` function, should be presented in reverse order.
- **multiple_values** (*bool*) – Whether more than one instance of the `rdftype` value should be processed (such as multiple keywords each specified by one `dcterms:subject` triple).
- **dimension_type** (*str*) – The general type of this facet – can be `"type"` (values are `rdf:type`), `"ref"` (values are URIs), `"year"` (values are xsd:datetime or similar), or `"value"` (values are string literals).
- **dimension_label** (*str*) – An alternate label for this facet to be used if the `selector` logic is more transformative than selectional (ie. if it transforms dates to True or False values depending on whether they're April 1st, you might set this to "aprilfirst")
- **identificator** (*callable*) – A function that takes *(row, binding, resource_graph)* and returns an identifier-like string usable as an id string or URL segment.

If optional parameters aren't provided, then appropriate values are selected if rdfrtype is one of some common rdf properties:

facet	description
rdf:type	Grouped by `qname()` of the `rdf:type` of the document, eg. `foaf:Document`. Not used for toc
dcterms:title	Grouped by first "sortable" letter, eg for a document titled "The Little Prince" returns "l". Is used as a facet for the API, but it's debatable if it's useful
dcterms:identifier	Also grouped by first sortable letter. When indexing, the resulting fulltext index field has a high boost value, which increases the chances of this document ranking high when one searches for its identifier.
dcterms:abstract	Not used for toc
dc:creator	Should be a free-test (string literal) value
dcterms:publisher	Should be a URIRef
dcterms:references	
dcterms:issued	Used for grouping documents published/issued in the same year
dc:subject	A document can have multiple dc:subjects and all are indexed/processed
dcterms:subject	Works like dc:subject, but the value should be a URIRef
schema:free	A boolean value

This module contains a number of classmethods that can be used as arguments to `selector` and `key`, eg

```
>>> from rdflib import Namespace
>>> MYVOCAB = Namespace("http://example.org/vocab/")
>>> f = Facet(MYVOCAB.enactmentDate, selector=Facet.year)
>>> f.selector({'myvocab_enactmentDate': '2014-07-06'},
...            'myvocab_enactmentDate')
'2014'
```

classmethod `defaultselector`(*row*, *binding*, *resource_graph=None*)

This returns `row[binding]` without any transformation.

```
>>> row = {"rdf_type": "http://purl.org/ontology/bibo/Book",
...        "dcterms_title": "A Tale of Two Cities",
...        "dcterms_issued": "1859-04-30",
...        "dcterms_publisher": "http://example.org/chapman_hall",
...        "schema_free": "true"}
>>> Facet.defaultselector(row, "dcterms_title")
'A Tale of Two Cities'
```

classmethod `year`(*row*, *binding=u'dcterms_issued'*, *resource_graph=None*)

This returns the the year part of `row[binding]`.

```
>>> row = {"rdf_type": "http://purl.org/ontology/bibo/Book",
...        "dcterms_title": "A Tale of Two Cities",
...        "dcterms_issued": "1859-04-30",
...        "dcterms_publisher": "http://example.org/chapman_hall",
...        "schema_free": "true"}
>>> Facet.year(row, "dcterms_issued")
'1859'
```

classmethod `booleanvalue`(*row*, *binding=u'schema_free'*, *resource_graph=None*)

Returns True iff row[binding] == "true", False otherwise.

```
>>> row = {"rdf_type": "http://purl.org/ontology/bibo/Book",
...        "dcterms_title": "A Tale of Two Cities",
```

```
...         "dcterms_issued": "1859-04-30",
...         "dcterms_publisher": "http://example.org/chapman_hall",
...         "schema_free": "true"}
>>> Facet.booleanvalue(row, "schema_free")
True
```

classmethod titlesortkey(*row*, *binding=u'dcterms_title'*, *resource_graph=None*)

Returns a version of row[binding] suitable for sorting. The function `title_sortkey()` is used for string transformation.

```
>>> row = {"rdf_type": "http://purl.org/ontology/bibo/Book",
...         "dcterms_title": "A Tale of Two Cities",
...         "dcterms_issued": "1859-04-30",
...         "dcterms_publisher": "http://example.org/chapman_hall",
...         "schema_free": "true"}
>>> Facet.titlesortkey(row, "dcterms_title")
'ataleoftwocities'
```

classmethod firstletter(*row*, *binding=u'dcterms_title'*, *resource_graph=None*)

Returns the first letter of row[binding], transformed into a sortable string.

```
>>> row = {"rdf_type": "http://purl.org/ontology/bibo/Book",
...         "dcterms_title": "A Tale of Two Cities",
...         "dcterms_issued": "1859-04-30",
...         "dcterms_publisher": "http://example.org/chapman_hall",
...         "schema_free": "true"}
>>> Facet.firstletter(row, "dcterms_title")
'a'
```

classmethod resourcelabel(*row*, *binding=u'dcterms_publisher'*, *resource_graph=None*)

Lookup a suitable text label for row[binding] in resource_graph.

```
>>> row = {"rdf_type": "http://purl.org/ontology/bibo/Book",
...         "dcterms_title": "A Tale of Two Cities",
...         "dcterms_issued": "1859-04-30",
...         "dcterms_publisher": "http://example.org/chapman_hall",
...         "schema_free": "true"}
>>> import rdflib
>>> resources = rdflib.Graph().parse(format="turtle", data="""
... @prefix foaf: <http://xmlns.com/foaf/0.1/> .
...
... <http://example.org/chapman_hall> a foaf:Organization;
...     foaf:name "Chapman & Hall" .
...
... """)
>>> Facet.resourcelabel(row, "dcterms_publisher", resources)
'Chapman & Hall'
```

classmethod sortresource(*row*, *binding=u'dcterms_publisher'*, *resource_graph=None*)

Returns a sortable version of the resource label for `row[binding]`.

```
>>> row = {"rdf_type": "http://purl.org/ontology/bibo/Book",
...         "dcterms_title": "A Tale of Two Cities",
...         "dcterms_issued": "1859-04-30",
...         "dcterms_publisher": "http://example.org/chapman_hall",
...         "schema_free": "true"}
>>> import rdflib
>>> resources = rdflib.Graph().parse(format="turtle", data="""
... @prefix foaf: <http://xmlns.com/foaf/0.1/> .
```

```
...
... <http://example.org/chapman_hall> a foaf:Organization;
...     foaf:name "Chapman & Hall" .
...
... """)
>>> Facet.sortresource(row, "dcterms_publisher", resources)
'chapmanhall'
```

classmethod term(*row*, *binding=u'dcterms_publisher'*, *resource_graph=None*)
Returns the leaf part of the URI found in `row[binding]`.

```
>>> row = {"rdf_type": "http://purl.org/ontology/bibo/Book",
...        "dcterms_title": "A Tale of Two Cities",
...        "dcterms_issued": "1859-04-30",
...        "dcterms_publisher": "http://example.org/chapman_hall",
...        "schema_free": "true"}
>>> Facet.term(row, "dcterms_publisher")
'chapman_hall'
```

classmethod qname(*row*, *binding=u'rdf_type'*, *resource_graph=None*)
Returns the qname of the rdf URIref contained in row[binding], as determined by the namespace prefixes registered in resource_graph.

```
>>> row = {"rdf_type": "http://purl.org/ontology/bibo/Book",
...        "dcterms_title": "A Tale of Two Cities",
...        "dcterms_issued": "1859-04-30",
...        "dcterms_publisher": "http://example.org/chapman_hall",
...        "schema_free": "true"}
>>> import rdflib
>>> resources = rdflib.Graph()
>>> resources.bind("bibo", "http://purl.org/ontology/bibo/")
>>> Facet.qname(row, "rdf_type", resources)
'bibo:Book'
```

16.1.6 The `TocPage` class

class `ferenda.`**`TocPage`**(*linktext*, *title*, *binding*, *value*)
Represents a particular TOC page.

Parameters

- **linktext** – The text used for TOC links *to* this page, like "a" or "2013".
- **linktext** – str
- **label** (*str*) – A description of this page, like "Documents starting with 'a"'
- **binding** (*str*) – The variable binding used for defining this TOC page, like "title" or "issued"
- **value** (*str*) – The particular value of bound variable that corresponds to this TOC page, like "a" or "2013". The `selector` function of a `Facet` object is used to select this value out of the raw data.

16.1.7 The `TocPageset` class

class `ferenda.`**`TocPageset`**(*label*, *pages*, *predicate=None*)
Represents a particular set of TOC pages, structured around some particular attribute(s) of documents, like title

or publication date. `toc_pagesets()` returns a list of these objects, override that method to provide custom TocPageset objects.

param label A description of this set of TOC pages, like "By publication year"

type label str

param pages The set of `TocPage` objects that makes up this page set.

type pages list

param predicate The RDFLib predicate (if any) that this pageset is keyed on.

16.1.8 The `NewsCriteria` class

class `ferenda.`**`NewsCriteria`**(*basefile*, *feedtitle*, *selector=None*, *key=None*)

Represents a particular subset of the documents in a repository, for the purpose of generating a news feed for that subset. These criteria objects are used by `news()`, and the simplest way of controlling which criteria are used for your docrepo is to override `news_criteria()` to make it return a list of instantiated NewsCriteria objects.

Parameters

- **basefile** (*str*) – A slug-like basic text label for this subset.
- **feedtitle** (*str*) – The title for this particular news feed
- **selector** (*callable*) – Function that takes a single `DocumentEntry` object and returns true iff it should be included in this feed.
- **key** (*callable*) – Function that takes a single `DocumentEntry` object and returns a value that can be used for sorting that object. The default implementation returns the entrys `updated` attribute, so that the feed contains entries sorted most recently updated first.

16.1.9 The `LayeredConfig` class

class `ferenda.`**`LayeredConfig`**(*defaults=None*, *inifile=None*, *commandline=None*, *cascade=False*)

Provides unified access to a nested set of configuration parameters. The source of these parameters a config file (using .ini-file style syntax), command line parameters, and default settings embedded in code. Command line parameters override configuration file parameters, which in turn override default settings in code (hence **Layered** Config).

Configuration parameters are accessed as regular object attributes, not dict-style key/value pairs. Configuration parameter names should therefore be regular python identifiers (i.e. only consist of alphanumerical chars and '_').

Configuration parameter values can be typed (strings, integers, booleans, dates, lists...). Even though ini-style config files and command line parameters are by themselves non-typed, by specifying default settings in code, parameters from a config file or the commamd line can be typed.

Configuration parameters can be changed in code. Such changes can be written to the configuration file by calling `write()`.

Parameters

- **defaults** (*dict*) – A dict with configuration keys and values. If any values are dicts, these are turned into nested config objects.

- **inifile** (*str*) – The name of a ini-style configuration file. The file should have a top-level section named __root__, whose keys are turned into top-level configuration parameters. Any other sections in this file are turned into nested config objects.
- **inifile** – full path to a .ini-style config file.
- **commandline** (*list*) – The contents of sys.argv, or something similar. Any long-style parameters are turned into configuration values, and parameters with hyphens are turned into nested config objects (i.e. `--module-parameter=foo` results in self.module.parameter == "foo".
- **cascade** (*bool*) – If an attempt to get a non-existing parameter on a sub (nested) configuration object should attempt to get the parameter on the parent config object.
- **cascade** – If a configuration key is not found, search parent config object.

Example:

```
>>> defaults = {'parameter': 'foo', 'other': 'default'}
>>> dir = tempfile.mkdtemp()
>>> inifile = dir + os.sep + "test.ini"
>>> with open(inifile, "w") as fp:
...     res = fp.write("[__root__]\nparameter = bar")
>>> argv = ['--parameter=baz']
>>> conf = LayeredConfig(defaults, inifile, argv)
>>> conf.parameter == 'baz'
True
>>> conf.other == 'default'
True
>>> conf.parameter = 'changed'
>>> conf.other = 'also changed'
>>> LayeredConfig.write(conf)
>>> with open(inifile) as fp:
...     res = fp.read()
>>> res == '[__root__]\nparameter = changed\nother = also changed\n\n'
True
>>> os.unlink(inifile)
>>> os.rmdir(dir)
```

static `write`(*config*)

Write changed properties to inifile (if provided at initialization).

16.1.10 The `elements` classes

16.1.11 The `elements.html` classes

The purpose of this module is to provide classes corresponding to most elements (except `<style>`, `<script>` and similar non-document content elements) and core attributes (except `@style` and the `%events` attributes) of HTML4.01 and HTML5. It is not totally compliant with the HTML4.01 and HTML5 standards, but is enough to model most real-world HTML. It contains no provisions to ensure that elements of a particular kind only contain allowed sub-elements.

`ferenda.elements.html.`**`elements_from_soup`**(*soup*, *remove_tags=(u'script', u'style', u'font', u'map', u'center')*, *keep_attributes=(u'class', u'id', u'dir', u'lang', u'src', u'href', u'name', u'alt')*)

Converts a BeautifulSoup tree into a tree of `ferenda.elements.html.HTMLElement` objects. Some non-semantic attributes and tags are removed in the process.

Parameters

- **soup** – Soup object to convert
- **remove_tags** (*list*) – Tags that should not be included
- **keep_attributes** (*list*) – Attributes to keep

Returns tree of element objects

Return type ferenda.elements.html.HTMLElement

class ferenda.elements.html.**HTMLElement**(**args*, ***kwargs*)
Abstract base class for all elements.

class ferenda.elements.html.**HTML**(**args*, ***kwargs*)
Element corresponding to the <html> tag

class ferenda.elements.html.**Head**(**args*, ***kwargs*)
Element corresponding to the <head> tag

class ferenda.elements.html.**Title**(**args*, ***kwargs*)
Element corresponding to the <title> tag

class ferenda.elements.html.**Body**(**args*, ***kwargs*)
Element corresponding to the <body> tag

class ferenda.elements.html.**P**(**args*, ***kwargs*)
Element corresponding to the <p> tag

class ferenda.elements.html.**H1**(**args*, ***kwargs*)
Element corresponding to the <h1> tag

class ferenda.elements.html.**H2**(**args*, ***kwargs*)
Element corresponding to the <h2> tag

class ferenda.elements.html.**H3**(**args*, ***kwargs*)
Element corresponding to the <h3> tag

class ferenda.elements.html.**H4**(**args*, ***kwargs*)
Element corresponding to the <h4> tag

class ferenda.elements.html.**H5**(**args*, ***kwargs*)
Element corresponding to the <h5> tag

class ferenda.elements.html.**H6**(**args*, ***kwargs*)
Element corresponding to the <h6> tag

class ferenda.elements.html.**UL**(**args*, ***kwargs*)
Element corresponding to the <ul> tag

class ferenda.elements.html.**OL**(**args*, ***kwargs*)
Element corresponding to the <ol> tag

class ferenda.elements.html.**LI**(**args*, ***kwargs*)
Element corresponding to the <li> tag

class ferenda.elements.html.**Pre**(**args*, ***kwargs*)
Element corresponding to the <pre> tag

class ferenda.elements.html.**DL**(**args*, ***kwargs*)
Element corresponding to the <dl> tag

class ferenda.elements.html.**DT**(**args*, ***kwargs*)
Element corresponding to the <dt> tag

class ferenda.elements.html.**DD**(**args*, ***kwargs*)
Element corresponding to the <dd> tag

class ferenda.elements.html.**Div**(**args*, ***kwargs*)
Element corresponding to the tag

class ferenda.elements.html.**Blockquote**(**args*, ***kwargs*)
Element corresponding to the <blockquote> tag

class ferenda.elements.html.**Form**(**args*, ***kwargs*)
Element corresponding to the <form> tag

class ferenda.elements.html.**HR**(**args*, ***kwargs*)
Element corresponding to the <hr> tag

class ferenda.elements.html.**Table**(**args*, ***kwargs*)
Element corresponding to the <table> tag

class ferenda.elements.html.**Fieldset**(**args*, ***kwargs*)
Element corresponding to the <fieldset> tag

class ferenda.elements.html.**Address**(**args*, ***kwargs*)
Element corresponding to the <address> tag

class ferenda.elements.html.**TT**(**args*, ***kwargs*)
Element corresponding to the <tt > tag

class ferenda.elements.html.**I**(**args*, ***kwargs*)
Element corresponding to the <i > tag

class ferenda.elements.html.**B**(**args*, ***kwargs*)
Element corresponding to the <b > tag

class ferenda.elements.html.**U**(**args*, ***kwargs*)
Element corresponding to the <u > tag

class ferenda.elements.html.**Big**(**args*, ***kwargs*)
Element corresponding to the <big > tag

class ferenda.elements.html.**Small**(**args*, ***kwargs*)
Element corresponding to the <small> tag

class ferenda.elements.html.**Em**(**args*, ***kwargs*)
Element corresponding to the tag

class ferenda.elements.html.**Strong**(**args*, ***kwargs*)
Element corresponding to the tag

class ferenda.elements.html.**Dfn**(**args*, ***kwargs*)
Element corresponding to the <dfn > tag

class ferenda.elements.html.**Code**(**args*, ***kwargs*)
Element corresponding to the <code > tag

class ferenda.elements.html.**Samp**(**args*, ***kwargs*)
Element corresponding to the <samp > tag

class ferenda.elements.html.**Kbd**(**args*, ***kwargs*)
Element corresponding to the <kbd > tag

class ferenda.elements.html.**Var**(**args*, ***kwargs*)
Element corresponding to the <var > tag

class ferenda.elements.html.**Cite**(**args*, ***kwargs*)
Element corresponding to the <cite > tag

class ferenda.elements.html.**Abbr**(**args*, ***kwargs*)
Element corresponding to the <abbr > tag

class ferenda.elements.html.**Acronym**(**args*, ***kwargs*)
Element corresponding to the <acronym> tag

class ferenda.elements.html.**A**(**args*, ***kwargs*)
Element corresponding to the <a > tag

class ferenda.elements.html.**Img**(**args*, ***kwargs*)
Element corresponding to the <img > tag

class ferenda.elements.html.**Object**(**args*, ***kwargs*)
Element corresponding to the <object > tag

class ferenda.elements.html.**Br**(**args*, ***kwargs*)
Element corresponding to the
 tag

class ferenda.elements.html.**Q**(**args*, ***kwargs*)
Element corresponding to the <q > tag

class ferenda.elements.html.**Sub**(**args*, ***kwargs*)
Element corresponding to the <sub > tag

class ferenda.elements.html.**Sup**(**args*, ***kwargs*)
Element corresponding to the <sup > tag

class ferenda.elements.html.**Span**(**args*, ***kwargs*)
Element corresponding to the tag

class ferenda.elements.html.**BDO**(**args*, ***kwargs*)
Element corresponding to the <bdo> tag

class ferenda.elements.html.**Input**(**args*, ***kwargs*)
Element corresponding to the <input> tag

class ferenda.elements.html.**Select**(**args*, ***kwargs*)
Element corresponding to the <select> tag

class ferenda.elements.html.**Textarea**(**args*, ***kwargs*)
Element corresponding to the <textarea> tag

class ferenda.elements.html.**Label**(**args*, ***kwargs*)
Element corresponding to the <label> tag

class ferenda.elements.html.**Button**(**args*, ***kwargs*)
Element corresponding to the <button> tag

class ferenda.elements.html.**Caption**(**args*, ***kwargs*)
Element corresponding to the <caption> tag

class ferenda.elements.html.**Thead**(**args*, ***kwargs*)
Element corresponding to the <thead> tag

class ferenda.elements.html.**Tfoot**(**args*, ***kwargs*)
Element corresponding to the <tfoot> tag

class ferenda.elements.html.**Tbody**(**args*, ***kwargs*)
Element corresponding to the <tbody> tag

class ferenda.elements.html.**Colgroup**(**args*, ***kwargs*)
Element corresponding to the <colgroup> tag

class ferenda.elements.html.**Col**(**args*, ***kwargs*)
Element corresponding to the <col> tag

class ferenda.elements.html.**TR**(**args*, ***kwargs*)
Element corresponding to the <tr> tag

class ferenda.elements.html.**TH**(**args*, ***kwargs*)
Element corresponding to the <th> tag

class ferenda.elements.html.**TD**(**args*, ***kwargs*)
Element corresponding to the <td> tag

class ferenda.elements.html.**Ins**(**args*, ***kwargs*)
Element corresponding to the <ins> tag

class ferenda.elements.html.**Del**(**args*, ***kwargs*)
Element corresponding to the <del> tag

class ferenda.elements.html.**Article**(**args*, ***kwargs*)
Element corresponding to the <article> tag

class ferenda.elements.html.**Aside**(**args*, ***kwargs*)
Element corresponding to the <aside> tag

class ferenda.elements.html.**Bdi**(**args*, ***kwargs*)
Element corresponding to the <bdi> tag

class ferenda.elements.html.**Details**(**args*, ***kwargs*)
Element corresponding to the <details> tag

class ferenda.elements.html.**Dialog**(**args*, ***kwargs*)
Element corresponding to the <dialog> tag

class ferenda.elements.html.**Summary**(**args*, ***kwargs*)
Element corresponding to the <summary> tag

class ferenda.elements.html.**Figure**(**args*, ***kwargs*)
Element corresponding to the <figure> tag

class ferenda.elements.html.**Figcaption**(**args*, ***kwargs*)
Element corresponding to the <figcaption> tag

class ferenda.elements.html.**Footer**(**args*, ***kwargs*)
Element corresponding to the <footer> tag

class ferenda.elements.html.**Header**(**args*, ***kwargs*)
Element corresponding to the <header> tag

class ferenda.elements.html.**Hgroup**(**args*, ***kwargs*)
Element corresponding to the <hgroup> tag

class ferenda.elements.html.**Mark**(**args*, ***kwargs*)
Element corresponding to the <mark> tag

class ferenda.elements.html.**Meter**(**args*, ***kwargs*)
Element corresponding to the <meter> tag

class ferenda.elements.html.**Nav**(**args*, ***kwargs*)
Element corresponding to the <nav> tag

class ferenda.elements.html.**Progress**(**args*, ***kwargs*)
Element corresponding to the <progress> tag

class ferenda.elements.html.**Ruby**(**args*, ***kwargs*)
Element corresponding to the <ruby> tag

class ferenda.elements.html.**Rt**(**args*, ***kwargs*)
Element corresponding to the <rt> tag

class ferenda.elements.html.**Rp**(**args*, ***kwargs*)
Element corresponding to the <rp> tag

class ferenda.elements.html.**Section**(**args*, ***kwargs*)
Element corresponding to the <section> tag

class ferenda.elements.html.**Time**(**args*, ***kwargs*)
Element corresponding to the <time> tag

class ferenda.elements.html.**Wbr**(**args*, ***kwargs*)
Element corresponding to the <wbr> tag

16.1.12 The Describer class

class ferenda.**Describer**(*graph=None*, *about=None*, *base=None*)
Extends the utility class rdflib.extras.describer.Describer so that it reads values and refences as well as write them.

Parameters

- **graph** (Graph) – The graph to read from and write to
- **about** (string or Identifier) – the current subject to use
- **base** (*string*) – Base URI for any relative URIs used with about(), rel() or rev(),

getvalues(*p*)
Get a list (possibly empty) of all literal values for the given property and the current subject. Values will be converted to plain literals, i.e. not rdflib.term.Literal objects.

Parameters **p** (rdflib.term.URIRef) – The property of the sought literal.

Returns a list of matching literals

Return type list of strings (or other appropriate python type if the literal has a datatype)

getrels(*p*)
Get a list (possibly empty) of all URIs for the given property and the current subject. Values will be converted to strings, i.e. not rdflib.term.URIRef objects.

Parameters **p** (rdflib.term.URIRef) – The property of the sought URI.

Returns The matching URIs

Return type list of strings

getrdftype()
Get the *rdf:type* of the current subject.

Returns The URI of the current subjects's rdf:type.

Return type string

getvalue(*p*)
Get a single literal value for the given property and the current subject. If the graph contains zero or more than one such literal, a `KeyError` will be raised.

Note: If this is all you use `Describer` for, you might want to use `rdflib.graph.Graph.value()` instead – the main advantage that this method has is that it converts the return value to a plain python object instead of a `rdflib.term.Literal` object.

Parameters **p** (`rdflib.term.URIRef`) – The property of the sought literal.

Returns The sought literal

Return type string (or other appropriate python type if the literal has a datatype)

getrel(*p*)
Get a single URI for the given property and the current subject. If the graph contains zero or more than one such URI, a `KeyError` will be raised.

Parameters **p** (`rdflib.term.URIRef`) – The property of the sought literal.

Returns The sought URI

Return type string

about(*subject*, ***kws*)
Sets the current subject. Will convert the given object into an `URIRef` if it's not an `Identifier`.

Usage:

```
>>> d = Describer()
>>> d._current()
rdflib.term.BNode(...)
>>> d.about("http://example.org/")
>>> d._current()
rdflib.term.URIRef(u'http://example.org/')
```

rdftype(*t*)
Shorthand for setting rdf:type of the current subject.

Usage:

```
>>> from rdflib import URIRef
>>> from rdflib.namespace import RDF, RDFS
>>> d = Describer(about="http://example.org/")
>>> d.rdftype(RDFS.Resource)
>>> (URIRef('http://example.org/'),
...     RDF.type, RDFS.Resource) in d.graph
True
```

rel(*p*, *o=None*, ***kws*)
Set an object for the given property. Will convert the given object into an `URIRef` if it's not an `Identifier`. If none is given, a new `BNode` is used.

Returns a context manager for use in a `with` block, within which the given object is used as current subject.

Usage:

```
>>> from rdflib import URIRef
>>> from rdflib.namespace import RDF, RDFS
>>> d = Describer(about="/", base="http://example.org/")
```

```
>>> _ctxt = d.rel(RDFS.seeAlso, "/about")
>>> d.graph.value(URIRef('http://example.org/'), RDFS.seeAlso)
rdflib.term.URIRef(u'http://example.org/about')

>>> with d.rel(RDFS.seeAlso, "/more"):
...     d.value(RDFS.label, "More")
>>> (URIRef('http://example.org/'), RDFS.seeAlso,
...         URIRef('http://example.org/more')) in d.graph
True
>>> d.graph.value(URIRef('http://example.org/more'), RDFS.label)
rdflib.term.Literal(u'More')
```

rev(*p*, *s=None*, ***kws*)

Same as `rel`, but uses current subject as *object* of the relation. The given resource is still used as subject in the returned context manager.

Usage:

```
>>> from rdflib import URIRef
>>> from rdflib.namespace import RDF, RDFS
>>> d = Describer(about="http://example.org/")
>>> with d.rev(RDFS.seeAlso, "http://example.net/"):
...     d.value(RDFS.label, "Net")
>>> (URIRef('http://example.net/'), RDFS.seeAlso,
...         URIRef('http://example.org/')) in d.graph
True
>>> d.graph.value(URIRef('http://example.net/'), RDFS.label)
rdflib.term.Literal(u'Net')
```

value(*p*, *v*, ***kws*)

Set a literal value for the given property. Will cast the value to an `Literal` if a plain literal is given.

Usage:

```
>>> from rdflib import URIRef
>>> from rdflib.namespace import RDF, RDFS
>>> d = Describer(about="http://example.org/")
>>> d.value(RDFS.label, "Example")
>>> d.graph.value(URIRef('http://example.org/'), RDFS.label)
rdflib.term.Literal(u'Example')
```

16.1.13 The `Transformer` class

class `ferenda.`**`Transformer`**(*transformertype*, *template*, *templatedirs*, *documentroot=None*, *config=None*)

Transforms parsed "pure content" documents into "browser-ready" HTML5 files with site branding and navigation, using a template of some kind.

Parameters

- **transformertype** (*str*) – The engine to be used for transforming. Right now only `"XSLT"` is supported.
- **template** (*str*) – The main template file.
- **templatedirs** (*str*) – Directories that may contain supporting templates used by the main template.

- **documentroot** (*str*) – The base directory for all generated files – used to make relative references to CSS/JS files correct.
- **config** – Any configuration information used by the transforming engine. Can be a path to a config file, a python data structure, or anything else compatible with the engine selected by `transformertype`.

Note: An initialized Transformer object only transforms using the template file provided at initialization. If you need to use another template file, create another Transformer object.

---

transform(*indata*, *depth*, *parameters=None*, *uritransform=None*)

Perform the transformation. This method always operates on the "native" datastructure – this might be different depending on the transformer engine. For XSLT, which is implemented through lxml, its in- and outdata are lxml trees

If you need an engine-indepent API, use `transform_stream()` or `transform_file()` instead

Parameters

- **indata** – The document to be transformed
- **depth** (*int*) – The directory nesting level, compared to `documentroot`
- **parameters** (*dict*) – Any parameters that should be provided to the template
- **uritransform** (*callable*) – A function, when called with an URI, returns a transformed URI/URL (such as the relative path to a static file) – used when transforming to files used for static offline use.

Returns The transformed document

transform_stream(*instream*, *depth*, *parameters=None*, *uritransform=None*)

Accepts a file-like object, returns a file-like object.

transform_file(*infile*, *outfile*, *parameters=None*, *uritransform=None*)

Accepts two filenames, reads from *infile*, writes to *outfile*.

16.1.14 The `FSMParser` class

class `ferenda.`**`FSMParser`**

A configurable finite state machine (FSM) for parsing documents with nested structure. You provide a set of *recognizers*, a set of *constructors*, a *transition table* and a *stream* of document text chunks, and it returns a hierarchical document object structure.

See *Parsing document structure*.

set_recognizers(**args*)

Set the list of functions (or other callables) used in order to recognize symbols from the stream of text chunks. Recognizers are tried in the order specified here.

set_transitions(*transitions*)

Set the transition table for the state matchine.

Parameters **transitions** – The transition table, in the form of a mapping between two tuples. The first tuple should be the current state (or a list of possible current states) and a callable function that determines if a particular symbol is recognized (`currentstate, recognizer`). The second tuple should be a constructor function (or *False*‘‘) and the new state to transition into.

parse(*chunks*)
Parse a document in the form of an iterable of suitable chunks – often lines or elements. each chunk should be a string or a string-like obje ct. Some examples:

```
p = FSMParser()
reader = TextReader("foo.txt")
body = p.parse(reader.getiterator(reader.readparagraph),"body", make_body)
body = p.parse(BeautifulSoup("foo.html").find_all("#main p"), "body", make_body)
body = p.parse(ElementTree.parse("foo.xml").find(".//paragraph"), "body", make_body)
```

Parameters

- **chunks** – The document to be parsed, as a list or any other iterable of text-like objects.
- **initialstate** – The initial state for the machine. The state must be present in the transition table. This could be any object, but strings are preferrable as they make error messages easier to understand.
- **initialconstructor** (*callable*) – A function that creates a document root object, and then fills it with child objects using .make_children()

Returns A document object tree.

analyze_symbol()
Internal function used by make_children()

transition(*currentstate*, *symbol*)
Internal function used by make_children()

make_child(*constructor*, *childstate*)
Internal function used by make_children(), which calls one of the constructors defined in the transition table.

make_children(*parent*)
Creates child nodes for the current (parent) document node.

Parameters **parent** – The parent document node, as any list-like object (preferrably a subclass of `ferenda.elements.CompoundElement`)

Returns The same `parent` object.

16.1.15 The `CitationParser` class

class `ferenda.`**CitationParser**(**grammars*)
Finds citations to documents and other resources in text strings. Each type of citation is specified by a pyparsing grammar, and for each found citation a URI can be constructed using a `URIFormatter` object.

Parameters **grammars** (list of `pyparsing.ParserElement` objects) – The grammar(s) for the citations that this parser should find, in order of priority.

Usage:

```
>>> from pyparsing import Word,nums
>>> rfc_grammar = ("RFC " + Word(nums).setResultsName("rfcnumber")).setResultsName("rfccite")
>>> pep_grammar = ("PEP" +  Word(nums).setResultsName("pepnumber")).setResultsName("pepcite")
>>> citparser = CitationParser(rfc_grammar, pep_grammar)
>>> res = citparser.parse_string("The WSGI spec (PEP 333) references RFC 2616 (The HTTP spec)")
>>> # res is a list of strings and/or pyparsing.ParseResult objects
>>> from ferenda import URIFormatter
>>> from ferenda.elements import Link
```

```
>>> f = URIFormatter(('rfccite',
...                   lambda p: "http://www.rfc-editor.org/rfc/rfc%(rfcnumber)s" % p),
...                  ('pepcite',
...                   lambda p: "http://www.python.org/dev/peps/pep-0%(pepnumber)s/" % p))
>>> citparser.set_formatter(f)
>>> res = citparser.parse_recursive(["The WSGI spec (PEP 333) references RFC 2616 (The HTTP spec
>>> res == ['The WSGI spec (', Link('PEP 333',uri='http://www.python.org/dev/peps/pep-0333/'), '
True
```

set_formatter(*formatter*)
Specify how found citations are to be formatted when using `parse_recursive()`

Parameters **formatter** (`URIFormatter`) – The formatter object to use for all citations

add_grammar(*grammar*)
Add another grammar.

Parameters **grammar** (`pyparsing.ParserElement`) – The grammar to add

parse_string(*string*)
Find any citations in a text string, using the configured grammars.

Parameters **string** (*str*) – Text to parse for citations

Returns strings (for parts of the input text that do not contain any citation) and/or tuples (for found citation) consisting of (string, `pyparsing.ParseResult`)

Return type list

parse_recursive(*part*)
Traverse a nested tree of elements, finding citations in any strings contained in the tree. Found citations are marked up as `Link` elements with the uri constructed by the `URIFormatter` set by `set_formatter()`.

Parameters **part** (*list*) – The root element of the structure to parse

Returns a correspondingly nested structure.

Return type list

16.1.16 The `URIFormatter` class

class `ferenda.`**URIFormatter**(**formatters*)
Companion class to `ferenda.CitationParser`, that handles the work of formatting the dicts or dict-like objects that CitationParser creates.

The class is initialized with a list of formatters, where each formatter is a tuple (key, callable). When `format()` is passed a citation reference in the form of a `pyparsing.ParseResult` object (which has a `.getName` method), the name of that reference is matched against the key of all formatters. If there is a match, the corresponding callable is called with the parseresult object as a single parameter, and the resulting string is returned.

An initialized `URIFormatter` object is not used directly. Instead, call `ferenda.CitationParser.set_formatter()` with the object as parameter. See *Citation parsing*.

Parameters ***formatters** (*list*) – Formatters, each provided as a *(name, callable)* tuple.

format(*parseresult*)
Given a pyparsing.ParseResult object, finds a appropriate formatter for that result, and formats the result into a URI using that formatter.

addformatter(*key*, *func*)
Add a single formatter to the list of registered formatters after initialization.

formatterfor(*key*)
Returns an appropriate formatting callable for the given key, or None if not found.

16.1.17 The `TripleStore` class

class ferenda.**TripleStore**(*location*, *repository*, ***kwargs*)
Presents a limited but uniform interface to different triple stores. It supports both standalone servers accessed over HTTP (Fuseki and Sesame, right now) as well as RDFLib-based persistant stores (The SQLite and Sleepycat/BerkeleyDB backends are supported).

Note: This class does not implement the RDFlib store interface. Instead, it provides a small list of operations that is generally useful for the kinds of things that ferenda-based applications need to do.

This class is an abstract base class, and is not directly instantiated. Instead, call `connect()`, which returns an initialized object of the appropriate subclass. All subclasses implements the following API.

static connect(*storetype*, *location*, *repository*, ***kwargs*)
Returns a initialized object, the exact type depending on the `storetype` parameter.

Parameters

- **storetype** – The type of store to connect to (`"FUSEKI"`, `"SESAME"`, `"SLEEPYCAT"` or `"SQLITE"`)
- **location** – The URL or file path where the main repository is stored
- **repository** – The name of the repository to use with the main repository storage
- ****kwargs** – Any other named parameters are passed to the appropriate class constructor (see "Store-specific parameters" below).

Example:

```
>>> # creates a new SQLite db at /tmp/test.sqlite if not already present
>>> sqlitestore = TripleStore.connect("SQLITE", "/tmp/test.sqlite", "myrepo")
>>> sqlitestore.triple_count()
0
>>> sqlitestore.close()
>>> # connect to same db, but store all triples in memory (read-only)
>>> sqlitestore = TripleStore.connect("SQLITE", "/tmp/test.sqlite", "myrepo", inmemory=True)
>>> # connect to a remote Fuseki store over HTTP, using the command-line
>>> # tool curl for faster batch up/downloads
>>> fusekistore = TripleStore.connect("FUSEKI", "http://localhost:3030/", "ds", curl=True)
```

Store-specific parameters:

When using storetypes `SQLITE` or `SLEEPYCAT`, the `select()` and `construct()` methods can be sped up (around 150%) by loading the entire content of the triple store into memory, by setting the `inmemory` parameter to `True`

When using storetypes `FUSEKI` or `SESAME`, storage and retrieval of a large number of triples (particularly the `add_serialized_file()` and `get_serialized_file()` methods) can be sped up by setting the `curl` parameter to `True`, if the command-line tool curl is available.

add_serialized(*data*, *format*, *context=None*)
Add the serialized RDF statements in the string *data* directly to the repository.

add_serialized_file(*filename*, *format*, *context=None*)
Add the serialized RDF statements contained in the file *filename* directly to the repository.

get_serialized(*format=u'nt'*, *context=None*)
Returns a string containing all statements in the store, serialized in the selected format. Returns byte string, not unicode array!

get_serialized_file(*filename*, *format=u'nt'*, *context=None*)
Saves all statements in the store to *filename*.

select(*query*, *format=u'sparql'*)
Run a SPARQL SELECT query against the triple store and returns the results.

Parameters

- **query** (*str*) – A SPARQL query with all neccessary prefixes defined.
- **format** (*str*) – Either one of the standard formats for queries (`"sparql"`, `"json"` or `"binary"`) – returns whatever `requests.get().content` returns – or the special value `"python"` which returns a python list of dicts representing rows and columns.

construct(*query*)
Run a SPARQL CONSTRUCT query against the triple store and returns the results as a RDFLib graph

Parameters **query** (*str*) – A SPARQL query with all neccessary prefixes defined.

update(*query*)
Run a SPARQL UPDATE (or DELETE/DROP/CLEAR) against the triplestore. Returns nothing but may raise an exception if something went wrong.

Parameters **query** (*str*) – A SPARQL query with all neccessary prefixes defined.

triple_count(*context=None*)
Returns the number of triples in the repository.

clear(*context=None*)
Removes all statements from the repository (without removing the repository as such).

close()
Close all connections to the triplestore. Needed if using RDFLib-based triple store, a no-op if using HTTP based stores.

16.1.18 The `FulltextIndex` class

Abstracts access to full text indexes (right now only Whoosh and ElasticSearch is supported, but maybe later, Solr, Xapian and/or Sphinx will be supported).

class ferenda.**FulltextIndex**(*location*, *repos*)
This is the abstract base class for a fulltext index. You use it by calling the static method FulltextIndex.connect, passing a string representing the underlying fulltext engine you wish to use. It returns a subclass on which you then call further methods.

static connect(*indextype*, *location*, *repos*)
Open a fulltext index (creating it if it doesn't already exists).

Parameters

- **location** (*str*) – Type of fulltext index ("WHOOSH" or "ELASTICSEARCH")
- **location** – The file path of the fulltext index.

make_schema(*repos*)

get_default_schema()

exists()
Whether the fulltext index exists.

create(*repos*)
Creates a fulltext index using the provided schema.

destroy()
Destroys the index, if created.

open()
Opens the index so that it can be queried.

schema()
Returns the schema that actually is in use. A schema is a dict where the keys are field names and the values are any subclass of `ferenda.fulltextindex.IndexedType`

update(*uri*, *repo*, *basefile*, *text*, ***kwargs*)
Insert (or update) a resource in the fulltext index. A resource may be an entire document, but it can also be any part of a document that is referenceable (i.e. a document node that has `@typeof` and `@about` attributes). A document with 100 sections can be stored as 100 independent resources, as long as each section has a unique key in the form of a URI.

Parameters

- **uri** (*str*) – URI for the resource
- **repo** (*str*) – The alias for the document repository that the resource is part of
- **basefile** (*str*) – The basefile which contains resource
- **title** (*str*) – User-displayable title of resource (if applicable). Should not contain the same information as `identifier`.
- **identifier** (*str*) – User-displayable short identifier for resource (if applicable)

Note: Calling this method may not directly update the fulltext index – you need to call `commit()` or `close()` for that.

---

commit()
Commit all pending updates to the fulltext index.

close()
Commits all pending updates and closes the index.

doccount()
Returns the number of currently indexed (non-deleted) documents.

query(*q=None*, *pagenum=1*, *pagelen=10*, ***kwargs*)

Perform a free text query against the full text index, optionally restricted with parameters for individual fields.

Parameters

- **q** (*str*) – Free text query, using the selected full text index's prefered query syntax
- ****kwargs** (*dict*) – any parameter will be used to match a similarly-named field

Returns matching documents, each document as a dict of fields

Return type list

Note: The *kwargs* parameters do not yet do anything – only simple full text queries are possible.

fieldmapping = ()
A tuple of `(abstractfield, nativefield)` tuples. Each `abstractfield` should be a instance of a IndexedType-derived class. Each `nativefield` should be whatever kind of object that is used with the native fullltextindex API.

The methods `to_native_field()` and `from_native_field()` uses this tuple of tuples to convert fields.

to_native_field(*fieldobject*)
Given a abstract field (an instance of a IndexedType-derived class), convert to the corresponding native type for the fulltextindex in use.

from_native_field(*fieldobject*)
Given a fulltextindex native type, convert to the corresponding IndexedType object.

Datatype field classes

class ferenda.fulltextindex.**IndexedType**(***kwargs*)
Base class for a fulltext searchengine-independent representation of indexed data. By using IndexType-derived classes to represent the schema, it becomes possible to switch out search engines without affecting the rest of the code.

class ferenda.fulltextindex.**Identifier**(***kwargs*)
An identifier is a string, normally in the form of a URI, which uniquely identifies an indexed document.

class ferenda.fulltextindex.**Datetime**(***kwargs*)

class ferenda.fulltextindex.**Text**(***kwargs*)

class ferenda.fulltextindex.**Label**(***kwargs*)

class ferenda.fulltextindex.**Keyword**(***kwargs*)
A keyword is a single string from a controlled vocabulary.

class ferenda.fulltextindex.**Boolean**(***kwargs*)

class ferenda.fulltextindex.**URI**(***kwargs*)
Any URI (except the URI that identifies a indexed document – use Identifier for that).

class ferenda.fulltextindex.**Resource**(***kwargs*)
A fulltextindex.Resource is a URI that also has a human-readable label.

Search field classes

class ferenda.fulltextindex.**SearchModifier**(**values*)

class ferenda.fulltextindex.**Less**(*max*)

class ferenda.fulltextindex.**More**(*min*)

class ferenda.fulltextindex.**Between**(*min*, *max*)

16.1.19 The `TextReader` class

class `ferenda.`**`TextReader`**(*filename=None*, *encoding=None*, *string=None*, *linesep=None*)

Fancy file-like-class for reading (not writing) text files by line, paragraph, page or any other user-defined unit of text, with support for peeking ahead and looking backwards. It can read files with byte streams using different encodings, but converts/handles everything to real strings (unicode in python 2). Alternatively, it can be initialized from an existing string.

Parameters

- **filename** (*str*) – The file to read
- **encoding** (*str*) – The encoding used by the file (default `ascii`)
- **string** (*str*) – Alternatively, a string used for initialization
- **linesep** (*str*) – The line separators used in the file/string

`UNIX` = u'\n'

Unix line endings, for use with the `linesep` parameter.

`DOS` = u'\r\n'

Dos/Windows line endings, for use with the `linesep` parameter.

`MAC` = u'\r'

Old-style Mac line endings, for use with the `linesep` parameter.

`eof`()

Returns True iff current seek position is at end of file.

`bof`()

Returns True iff current seek position is at begining of file.

`cue`(*string*)

Set seek position at the beginning of *string*, starting at current seek position. Raises IOError if *string* not found.

`cuepast`(*string*)

Set seek position at the beginning of *string*, starting at current seek position. Raises IOError if *string* not found.

`readto`(*string*)

Read and return all text between current seek potition and *string*. Sets new seek position at the start of *string*. Raises IOError if *string* not found.

`readparagraph`()

Reads and returns the next paragraph (all text up to two or more consecutive line separators).

`readpage`()

Reads and returns the next page (all text up to next form feed, `"\f"`)

`readchunk`(*delimiter*)

Reads and returns the next chunk of text up to *delimiter*

`lastread`()

Returns the last chunk of data that was actually read (i.e. the `peek*` and `prev*` methods do not affect this)

`peek`(*size=0*)

Works like `read()`, but does not affect current seek position.

peekline(*times=1*)
Works like `readline()`, but does not affect current seek position. If *times* is specified, peeks that many lines ahead.

peekparagraph(*times=1*)
Works like `readparagraph()`, but does not affect current seek position. If *times* is specified, peeks that many paragraphs ahead.

peekchunk(*delimiter*, *times=1*)
Works like `readchunk()`, but does not affect current seek position. If *times* is specified, peeks that many chunks ahead.

prev(*size=0*)
Works like `read()`, but reads backwards from current seek position, and does not affect it.

prevline(*times=1*)
Works like `readline()`, but reads backwards from current seek position, and does not affect it. If *times* is specified, reads the line that many times back.

prevparagraph(*times=1*)
Works like `readparagraph()`, but reads backwards from current seek position, and does not affect it. If *times* is specified, reads the paragraph that many times back.

prevchunk(*delimiter*, *times=1*)
Works like `readchunk()`, but reads backwards from current seek position, and does not affect it. If *times* is specified, reads the chunk that many times back.

getreader(*callableObj*, **args*, ***kwargs*)
Enables you to treat the result of any single `read*`, `peek*` or `prev*` methods as a new TextReader. Particularly useful to process individual pages in page-oriented documents:

```
filereader = TextReader("rfc822.txt")
firstpagereader = filereader.getreader(filereader.readpage)
# firstpagereader is now a standalone TextReader that only
# contains the first page of text from rfc822.txt
filereader.seek(0) # reset current seek position
page5reader = filereader.getreader(filereader.peekpage, times=5)
# page5reader now contains the 5th page of text from rfc822.txt
```

getiterator(*callableObj*, **args*, ***kwargs*)
Returns an iterator:

```
filereader = TextReader("dashed.txt")
# dashed.txt contains paragraphs separated by "----"
for para in filereader.getiterator(filereader.readchunk, "----"):
    print(para)
```

flush()
See `io.IOBase.flush()`. This is a no-op.

read(*size=0*)
See `io.TextIOBase.read()`.

readline(*size=None*)
See `io.TextIOBase.readline()`.

Note: The `size` parameter is not supported.

---

seek(*offset*, *whence=0*)
See `io.TextIOBase.seek()`.

Note: The `whence` parameter is not supported.

tell()
See `io.TextIOBase.tell()`.

write(*str*)
See `io.TextIOBase.write()`.

Note: Always raises IOError, as TextReader is a read-only object.

writelines(*sequence*)
See `io.IOBase.writelines()`.

Note: Always raises IOError, as TextReader is a read-only object.

next()
Backwards-compatibility alias for iterating over a file in python 2. Use `getiterator()` to make iteration work over anything other than lines (eg paragraphs, pages, etc).

16.1.20 The `PDFReader` class

class `ferenda.`**PDFReader**(**args*, ***kwargs*)
Parses PDF files and makes the content available as a object hierarchy. After calling `read()`, the PDFReader itself is a list of `ferenda.pdfreader.Page` objects, which each is a list of `ferenda.pdfreader.Textbox` objects, which each is a list of `ferenda.pdfreader.Textelement` objects.

Note: This class depends on the command line tool pdftohtml from poppler.

The class can also handle any other type of document (such as Word/OOXML/WordPerfect/RTF) that OpenOffice or LibreOffice handles by first converting it to PDF using the `soffice` command line tool (which then must be in your `$PATH`).

If the PDF contains only scanned pages (without any OCR information), the pages can be run through the `tesseract` command line tool (which, again, needs to be in your `$PATH`). You need to provide the main language of the document as the `ocr_lang` parameter, and you need to have installed the tesseract language files for that language.

tagname = u'div'

classname = u'pdfreader'

read(*pdffile*, *workdir*, *images=True*, *convert_to_pdf=False*, *keep_xml=True*, *ocr_lang=None*)
Initializes a PDFReader object from an existing PDF file. After initialization, the PDFReader contains a list of `Page` objects.

Parameters

- **pdffile** – The full path to the PDF file (or, if `convert_to_pdf` is set, any other document file)
- **workdir** – A directory where intermediate files (particularly background PNG files) are stored

- **convert_to_pdf** (*bool*) – If pdffile is any other type of document other than PDF, attempt to first convert it to PDF using the `soffice` command line tool (from OpenOffice/LibreOffice).
- **keep_xml** (*bool*) – If False, remove the intermediate XML representation of the PDF that gets created in `workdir`. If true, keep it around to speed up subsequent parsing operations. If set to the special value `"bz2"`, keep it but compress it with `bz2`.
- **ocr_lang** – If provided, PDFReader will extract scanned images from the PDF file, and run an OCR program on it, using the `ocr_lang` language heuristics. (Note that this is not neccessarily an IETF language tag like "sv" or "en-GB", but rather whatever the underlying `tesseract` program uses).
- **ocr_lang** – str

is_empty()

textboxes(*gluefunc=None*, *pageobjects=False*, *keepempty=False*)
Return an iterator of the textboxes available.

`gluefunc` should be a callable that is called with (textbox, nextbox, prevbox), and returns True iff nextbox should be appended to textbox.

If `pageobjects`, the iterator can return Page objects to signal that pagebreak has ocurred (these Page objects may or may not have Textbox elements).

If `keepempty`, process and return textboxes that have no text content (these are filtered out by default)

drawboxes(*outfile*, *gluefunc=None*)
Create a copy of the parsed PDF file, but with the textboxes created by `gluefunc` clearly marked. Returns the name of the created pdf file.

..note:

```
This requires PyPDF2 and reportlab, which aren't installed
by default (and at least reportlab is not py3 compatible).
```

static **re_dimensions**()
search(string[, pos[, endpos]]) –> match object or None. Scan through string looking for a match, and return a corresponding match object instance. Return None if no position in the string matches.

median_box_width(*threshold=0*)
Returns the median box width of all pages.

class `ferenda.pdfreader.`**Page**(**args*, ***kwargs*)
Represents a Page in a PDF file. Has *width* and *height* properties.

tagname = u'div'

classname = u'pdfpage'

id

boundingbox(*top=0*, *left=0*, *bottom=None*, *right=None*)
A generator of `ferenda.pdfreader.Textbox` objects that fit into the bounding box specified by the parameters.

crop(*top=0*, *left=0*, *bottom=None*, *right=None*)
Removes any `ferenda.pdfreader.Textbox` objects that does not fit within the bounding box specified by the parameters.

class `ferenda.pdfreader.`**Textbox**(**args*, ***kwargs*)
A textbox is a amount of text on a PDF page, with *top*, *left*, *width* and *height* properties that specifies the bounding box of the text. The *font* property specifies the id of font used (use `getfont()` to get a dict of all

font properties). A textbox consists of a list of Textelements which may differ in basic formatting (bold and or italics), but otherwise all text in a Textbox has the same font and size.

tagname = u'p'

classname = u'textbox'

as_xhtml(*uri*)

getfont()
Returns a fontspec dict of all properties of the font used.

class ferenda.pdfreader.**Textelement**(**args*, ***kwargs*)
Represent a single part of text where each letter has the exact same formatting. The `tag` property specifies whether the text as a whole is bold (`'b'`) , italic(`'i'` bold + italic (`'bi'`) or regular (`None`).

tagname

16.1.21 The `WordReader` class

class ferenda.**WordReader**
Reads .docx and .doc-files (the latter with support from antiword) and presents a slightly easier API for dealing with them.

read(*wordfile*, *intermediatefile*)
Converts the word file to a more easily parsed format.

Parameters

- **wordfile** – Path to original docfile
- **intermediatefile** – Where to store the more parseable file

Returns name of parseable file, filetype (either "doc" or "docx")

Return type tuple

word_to_docbook(*indoc*, *outdoc*)
Convert a old Word document (.doc) to a pseudo-docbook file through antiword.

word_to_ooxml(*indoc*, *outdoc*)
Extracts the raw OOXML file from a modern Word document (.docx).

16.2 Modules

16.2.1 The `util` module

General library of small utility functions.

class ferenda.util.**gYearMonth**

class ferenda.util.**gYear**

ferenda.util.**ns**
A mapping of well-known prefixes and their corresponding namespaces. Includes `dc`, `dcterms`, `rdfs`, `rdf`, `skos`, `xsd`, `foaf`, `owl`, `xhv`, `prov` and `bibo`.

ferenda.util.**mkdir**(*newdir*)
Like `os.makedirs()`, but doesn't raise an exception if the directory already exists.

ferenda.util.**ensure_dir**(*filename*)
Given a filename (typically one that you wish to create), ensures that the directory the file is in actually exists.

ferenda.util.**robust_rename**(*old*, *new*)
Rename old to new no matter what (if the file exists, it's removed, if the target dir doesn't exist, it's created)

ferenda.util.**robust_remove**(*filename*)
Removes a filename no matter what (unlike `os.unlink()`, does not raise an error if the file does not exist).

ferenda.util.**relurl**(*url*, *starturl*)
Works like `os.path.relpath()`, but for urls

```
>>> relurl("http://example.org/other/index.html", "http://example.org/main/index.html") == '../o
True
>>> relurl("http://other.org/foo.html", "http://example.org/bar.html") == 'http://other.org/foo.
True
```

ferenda.util.**numcmp**(*x*, *y*)
Works like `cmp` in python 2, but compares two strings using a 'natural sort' order, ie "10" > "2". Also handles strings that contains a mixture of numbers and letters, ie "2" < "2 a".

Return negative if x<y, zero if x==y, positive if x>y.

```
>>> numcmp("10", "2")
1
>>> numcmp("2", "2 a")
-1
>>> numcmp("3", "2 a")
1
```

ferenda.util.**split_numalpha**(*s*)
Converts a string into a list of alternating string and integers. This makes it possible to sort a list of strings numerically even though they might not be fully convertable to integers

```
>>> split_numalpha('10 a §') == ['', 10, ' a §']
True
>>> sorted(['2 §', '10 §', '1 §'], key=split_numalpha) == ['1 §', '2 §', '10 §']
True
```

ferenda.util.**runcmd**(*cmdline*, *require_success=False*, *cwd=None*, *cmdline_encoding=None*, *output_encoding=u'utf-8'*)
Run a shell command, wait for it to finish and return the results.

Parameters

- **cmdline** (*str*) – The full command line (will be passed through a shell)
- **require_success** (*bool*) – If the command fails (non-zero exit code), raise `ExternalCommandError`
- **cwd** – The working directory for the process to run

Returns The returncode, all stdout output, all stderr output

Return type tuple

ferenda.util.**normalize_space**(*string*)
Normalize all whitespace in string so that only a single space between words is ever used, and that the string neither starts with nor ends with whitespace.

```
>>> normalize_space(" This is  a long \n string\n") == 'This is a long string'
True
```

ferenda.util.**list_dirs**(*d*, *suffix=None*, *reverse=False*)
A generator that works much like `os.listdir()`, only recursively (and only returns files, not directories).

Parameters

- **d** (*str*) – The directory to start in
- **suffix** (*str*) – Only return files with the given suffix
- **reverse** – Returns result sorted in reverse alphabetic order
- **type** –

Returns the full path (starting from d) of each matching file

Return type generator

ferenda.util.**replace_if_different**(*src*, *dst*, *archivefile=None*)
Like `shutil.move()`, except the *src* file isn't moved if the *dst* file already exists and is identical to *src*. Also doesn't require that the directory of *dst* exists beforehand.

Note: regardless of whether it was moved or not, *src* is always deleted.

Parameters

- **src** (*str*) – The source file to move
- **dst** (*str*) – The destination file

Returns True if src was copied to dst, False otherwise

Return type bool

ferenda.util.**copy_if_different**(*src*, *dest*)
Like `shutil.copyfile()`, except the *src* file isn't copied if the *dst* file already exists and is identical to *src*. Also doesn't require that the directory of *dst* exists beforehand.

param src The source file to move

type src str

param dst The destination file

type dst str

returns True if src was copied to dst, False otherwise

rtype bool

ferenda.util.**outfile_is_newer**(*infiles*, *outfile*)
Check if a given *outfile* is newer (has a more recent modification time) than a list of *infiles*. Returns True if so, False otherwise (including if outfile doesn't exist).

ferenda.util.**link_or_copy**(*src*, *dst*)
Create a symlink at *dst* pointing back to *src* on systems that support it. On other systems (i.e. Windows), copy *src* to *dst* (using `copy_if_different()`)

ferenda.util.**ucfirst**(*string*)
Returns string with first character uppercased but otherwise unchanged.

```
>>> ucfirst("iPhone") == 'IPhone'
True
```

ferenda.util.**rfc_3339_timestamp**(*dt*)
Converts a datetime object to a RFC 3339-style date

```
>>> rfc_3339_timestamp(datetime.datetime(2013, 7, 2, 21, 20, 25)) == '2013-07-02T21:20:25-00:00'
True
```

ferenda.util.**parse_rfc822_date**(*httpdate*)

Converts a RFC 822-type date string (more-or-less the same as a HTTP-date) to an UTC-localized (naive) datetime.

```
>>> parse_rfc822_date("Mon, 4 Aug 1997 02:14:00 EST")
datetime.datetime(1997, 8, 4, 7, 14)
```

ferenda.util.**strptime**(*datestr*, *format*)

Like datetime.strptime, but guaranteed to not be affected by current system locale – all datetime parsing is done using the C locale.

```
>>> strptime("Mon, 4 Aug 1997 02:14:05", "%a, %d %b %Y %H:%M:%S")
datetime.datetime(1997, 8, 4, 2, 14, 5)
```

ferenda.util.**readfile**(*filename*, *mode='r'*, *encoding='utf-8'*)

Opens *filename*, reads it's contents and returns them as a string.

ferenda.util.**writefile**(*filename*, *contents*, *encoding='utf-8'*)

Create *filename* and write *contents* to it.

ferenda.util.**extract_text**(*html*, *start*, *end*, *decode_entities=True*, *strip_tags=True*)

Given *html*, a string of HTML content, and two substrings (*start* and *end*) present in this string, return all text between the substrings, optionally decoding any HTML entities and removing HTML tags.

```
>>> extract_text("<body><div><b>Hello</b> <i>World</i>&trade;</div></body>",
...              "<div>", "</div>") == 'Hello World™'
True
>>> extract_text("<body><div><b>Hello</b> <i>World</i>&trade;</div></body>",
...              "<div>", "</div>", decode_entities=False) == 'Hello World&trade;'
True
>>> extract_text("<body><div><b>Hello</b> <i>World</i>&trade;</div></body>",
...              "<div>", "</div>", strip_tags=False) == '<b>Hello</b> <i>World</i>™'
True
```

ferenda.util.**merge_dict_recursive**(*base*, *other*)

Merges the *other* dict into the *base* dict. If any value in other is itself a dict and the base also has a dict for the same key, merge these sub-dicts (and so on, recursively).

```
>>> base = {'a': 1, 'b': {'c': 3}}
>>> other = {'x': 4, 'b': {'y': 5}}
>>> want = {'a': 1, 'x': 4, 'b': {'c': 3, 'y': 5}}
>>> got = merge_dict_recursive(base, other)
>>> got == want
True
>>> base == want
True
```

ferenda.util.**resource_extract**(*resource_name*, *outfile*, *params={}*)

Copy a file from the ferenda package resources to a specified path, optionally performing variable substitutions on the contents of the file.

Parameters

- **resource_name** – The named resource (eg 'res/sparql/annotations.rq')
- **outfile** – Path to extract the resource to

- **params** – A dict of parameters, to be used with regular string subtitutions in the resource file.

ferenda.util.**uri_leaf**(*uri*)

Get the "leaf" - fragment id or last segment - of a URI. Useful e.g. for getting a term from a "namespace like" URI.

```
>>> uri_leaf("http://purl.org/dc/terms/title") == 'title'
True
>>> uri_leaf("http://www.w3.org/2004/02/skos/core#Concept") == 'Concept'
True
>>> uri_leaf("http://www.w3.org/2004/02/skos/core#") # returns None
```

ferenda.util.**logtime**(**args*, ***kwds*)

A context manager that uses the supplied method and format string to log the elapsed time:

```
with util.logtime(log.debug,
                  "Basefile %(basefile)s took %(elapsed).3f s",
                  {'basefile':'foo'}):
    do_stuff_that_takes_some_time()
```

This results in a call like log.debug("Basefile foo took 1.324 s").

ferenda.util.**c_locale**(**args*, ***kwds*)

Temporarily change process locale to the C locale, for use when eg parsing English dates on a system that may have non-english locale.

```
>>> with c_locale():
...     datetime.datetime.strptime("August 2013", "%B %Y")
datetime.datetime(2013, 8, 1, 0, 0)
```

ferenda.util.**from_roman**(*s*)

convert Roman numeral to integer.

```
>>> from_roman("MCMLXXXIV")
1984
```

ferenda.util.**title_sortkey**(*s*)

Transform a document title into a key useful for sorting and partitioning documents.

```
>>> title_sortkey("The 'viewstate' property") == 'viewstateproperty'
True
```

ferenda.util.**parseresults_as_xml**(*parseres*, *depth=0*)

ferenda.util.**json_default_date**(*obj*)

16.2.2 The `citationpatterns` module

General ready-made grammars for use with `CitationParser`. See *Citation parsing* for examples.

ferenda.citationpatterns.**url**

Matchs any URL like 'http://example.com/ or 'https://example.org/?key=value#fragment' (note: only the schemes/protocols 'http', 'https' and 'ftp' are supported)

ferenda.citationpatterns.**eulaw**

Matches EU Legislation references like 'direktiv 2007/42/EU'.

16.2.3 The `uriformats` module

A small set of generic functions to convert (dicts or dict-like objects) to URIs. They are usually matched with a corresponding citationpattern like the ones found in `ferenda.citationpatterns`. See *Citation parsing* for examples.

`ferenda.uriformats.`**`generic`**(*d*)

Converts any dict into a URL. The domain (netlog) is always example.org, and all keys/values of the dict is turned into a querystring.

```
>>> generic({'foo':'1', 'bar':'2'})
"http://example.org/?foo=1&bar=2"
```

`ferenda.uriformats.`**`url`**(*d*)

Converts a dict with keys `scheme`, `netloc`, `path` (and optionally query and/or fragment) into the corresponding URL.

```
>>> url({'scheme':'https', 'netloc':'example.org', 'path':'test'})
"https://example.org/test
```

`ferenda.uriformats.`**`eulaw`**(*d*)

Converts a dict with keys like LegalactType, Directive, ArticleId (produced by `ferenda.citationpatterns.eulaw`) into a CELEX-based URI.

> **Note:** This is not yet implemented.

16.2.4 The `manager` module

Utility functions for running various ferenda tasks from the command line, including registering classes in the configuration file. If you're using the `DocumentRepository` API directly in your code, you'll probably only need `makeresources()`, `frontpage()` and possibly `setup_logger()`. If you're using the `ferenda-build.py` tool, you don't need to directly call any of these methods – `ferenda-build.py` calls `run()`, which calls everything else, for you.

`ferenda.manager.`**`makeresources`**(*repos*, *resourcedir=u'data/rsrc'*, *combine=False*, *cssfiles=[]*, *jsfiles=[]*, *imgfiles=[]*, *staticsite=False*, *legacyapi=False*, *sitename=u'MySite'*, *sitedescription=u'Just another Ferenda site'*, *url=u'http://localhost:8000/'*)

Creates the web assets/resources needed for the web app (concatenated and minified js/css files, resources.xml used by most XSLT stylesheets, etc).

Parameters

- **repos** (*list*) – The repositories to create resources for, as instantiated and configured docrepo objects
- **combine** (*bool*) – whether to combine and compact/minify CSS and JS files
- **resourcedir** (*str*) – where to put generated/copied resources

Returns All created/copied css, js and resources.xml files

Return type dict of lists

`ferenda.manager.`**`frontpage`**(*repos*, *path=u'data/index.html'*, *stylesheet=u'res/xsl/frontpage.xsl'*, *sitename=u'MySite'*, *staticsite=False*)

Create a suitable frontpage.

Parameters

- **repos** (*list*) – The repositories to list on the frontpage, as instantiated and configured docrepo objects
- **path** (*str*) – the filename to create.

ferenda.manager.**runserver**(*repos*, *port=8000*, *documentroot=u'data'*, *apiendpoint=u'/api/'*, *searchendpoint=u'/search/'*, *url=u'http://localhost:8000/'*, *indextype=u'WHOOSH'*, *indexlocation=u'data/whooshindex'*, *legacyapi=False*)

Starts up a internal webserver and runs the WSGI app (see `make_wsgi_app()`) using all the specified document repositories. Runs forever (or until interrupted by keyboard).

Parameters

- **repos** (*list*) – Object instances for the repositories that should be served over HTTP
- **port** (*int*) – The port to use
- **documentroot** (*str*) – The root document, used to locate files not directly handled by any repository
- **apiendpoint** (*str*) – The part of the URI space handled by the API functionality
- **searchendpoint** (*str*) – The part of the URI space handled by the search functionality

ferenda.manager.**make_wsgi_app**(*inifile=None*, ***kwargs*)

Creates a callable object that can act as a WSGI application by mod_wsgi, gunicorn, the built-in webserver, or any other WSGI-compliant webserver.

Parameters

- **inifile** (*str*) – The full path to a `ferenda.ini` configuration file
- ****kwargs** – Configuration values for the wsgi app (must include `documentroot`, `apiendpoint` and `searchendpoint`). Only used if `inifile` is not provided.

Returns A WSGI application

Return type callable

ferenda.manager.**setup_logger**(*level=u'INFO'*, *filename=None*, *logformat=u'%(asctime)s %(name)s %(levelname)s %(message)s'*, *datefmt=u'%H:%M:%S'*)

Sets up the logging facilities and creates the module-global log object as a root logger.

Parameters

- **name** (*str*) – The name of the logger (used in log messages)
- **level** (*str*) – 'DEBUG','INFO','WARNING','ERROR' or 'CRITICAL'
- **filename** (*str*) – The name of the file to log to. If None, log to stdout

ferenda.manager.**shutdown_logger**()

Shuts down the configured logger. In particular, closes any FileHandlers, which is needed on win32.

ferenda.manager.**run**(*argv*)

Runs a particular action for either a particular class or all enabled classes.

Parameters **argv** – a `sys.argv`-style list of strings specifying the class to load, the action to run, and additional parameters. The first parameter is either the name of the class-or-alias, or the special value "all", meaning all registered classes in turn. The second parameter is the action to run, or the special value "all" to run all actions in correct order. Remaining parameters are

either configuration parameters (if prefixed with `--`, e.g. `--loglevel=INFO`, or positional arguments to the specified action).

`ferenda.manager.`**`enable`**(*classname*)
Registers a class by creating a section for it in the configuration file (`ferenda.ini`). Returns the short-form alias for the class.

```
>>> enable("ferenda.DocumentRepository") == 'base'
True
>>> os.unlink("ferenda.ini")
```

Parameters **classname** (*str*) – The fully qualified name of the class

Returns The short-form alias for the class

Return type str

`ferenda.manager.`**`runsetup`**()
Runs `setup()` and exits with a non-zero status if setup failed in any way

Note: The `ferenda-setup` script that gets installed with ferenda is a tiny wrapper around this function.

`ferenda.manager.`**`setup`**(*argv=None*, *force=False*, *verbose=False*, *unattended=False*)
Creates a project, complete with configuration file and ferenda-build tool.

Checks to see that all required python modules and command line utilities are present. Also checks which triple store(s) are available and selects the best one (in order of preference: Sesame, Fuseki, RDFLib+Sleepycat, RDFLib+SQLite).

16.2.5 The `testutil` module

`unittest`-based classes and accompanying functions to create some types of ferenda-specific tests easier.

class `ferenda.testutil.`**`FerendaTestCase`**
Convenience class with extra AssertEqual methods. Note that even though this method provides `unittest.TestCase`-like assert methods, it does not derive from `TestCase`. When creating a test case that makes use of these methods, you need to inherit from both `TestCase` and this class, ie:

```
class MyTestcase(unittest.TestCase, ferenda.testutil.FerendaTestCase):
    def test_simple(self):
        self.assertEqualXML("<foo arg1='x' arg2='y'/>", "<foo arg2='y' arg1='x'/>")
```

`assertEqualGraphs`(*want*, *got*, *exact=True*)
Assert that two RDF graphs are identical (isomorphic).

Parameters

- **want** – The graph as expected, as an `Graph` object or the filename of a serialized graph
- **got** – The actual graph, as an `Graph` object or the filename of a serialized graph
- **exact** (*bool*) – Whether to require that the graphs are exactly alike (True) or only if all triples in want exists in got (False)

`assertAlmostEqualDatetime`(*datetime1*, *datetime2*, *delta=1*)
Assert that two datetime objects are reasonably equal.

Parameters

- **datetime1** (*datetime*) – The first datetime to compare

- **datetime2** (*datetime*) – The second datetime to compare
- **delta** (*int*) – How much the datetimes are allowed to differ, in seconds.

assertEqualXML(*want*, *got*, *namespace_aware=True*)
Assert that two xml trees are canonically identical.

Parameters

- **want** – The XML document as expected, as a string, byte string or ElementTree element
- **got** – The actual XML document, as a string, byte string or ElementTree element

assertEqualDirs(*want*, *got*, *suffix=None*, *filterdir=u'entries'*)
Assert that two directory trees contains identical files

Parameters

- **want** (*str*) – The expected directory tree
- **got** (*str*) – The actual directory tree
- **suffix** (*str*) – If given, only check files ending in suffix (otherwise check all the files
- **filterdir** – If given, don't compare the parts of the tree that starts with filterdir

class ferenda.testutil.**RepoTester**(*methodName='runTest'*)
A unittest.TestCase-based convenience class for creating file-based integration tests for an entire docrepo. To use this, you only need a very small amount of boilerplate code, and some files containing data to be downloaded or parsed. The actual tests are dynamically created from these files. The boilerplate can look something like this:

```
class TestRFC(RepoTester):
    repoclass = RFC  # the docrepo class to test
    docroot = os.path.dirname(__file__)+"/files/repo/rfc"

parametrize_repotester(TestRFC)
```

repoclass
The actual documentrepository class to be tested. Must be overridden when creating a testcase class.

alias of DocumentRepository

docroot = u'/tmp'
The location of test files to create tests from. Must be overridden when creating a testcase class

filename_to_basefile(*filename*)
Converts a test filename to a basefile. Default implementation attempts to find out basefile from the reposclass being tested (or rather it's documentstore), but returns a hard-coded basefile if it fails.

Parameters **filename** (*str*) – The test file

Returns Corresponding basefile

Return type str

ferenda.testutil.**parametrize**(*cls*, *template_method*, *name*, *params*, *wrapper=None*)
Creates a new test method on a TestCase class, which calls a specific template method with the given parameters (ie. a parametrized test). Given a testcase like this:

```
class MyTest(unittest.TestCase):
    def my_general_test(self, parameter):
        self.assertEqual(parameter, "hello")
```

and the following top-level initalization code:

```
parametrize(MyTest,MyTest.my_general_test, "test_one", ["hello"])
parametrize(MyTest,MyTest.my_general_test, "test_two", ["world"])
```

you end up with a test case class with two methods. Using e.g. `unittest discover` (or any other unittest-compatible test runner), the following should be the result:

```
test_one (test_parametric.MyTest) ... ok
test_two (test_parametric.MyTest) ... FAIL

======================================================================
FAIL: test_two (test_parametric.MyTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "./ferenda/testutil.py", line 365, in test_method
    template_method(self, *params)
  File "./test_parametric.py", line 6, in my_general_test
    self.assertEqual(parameter, "hello")
AssertionError: 'world' != 'hello'
- world
+ hello
```

Parameters

- **cls** – The `TestCase` class to add the parametrized test to.
- **template_method** – The method to use for parametrization
- **name** (*str*) – The name for the new test method
- **params** (*list*) – The parameter list (Note: keyword parameters are not supported)
- **wrapper** – A unittest decorator like `unittest.skip()` or `unittest.expectedFailure()`.
- **wrapper** – callable

`ferenda.testutil.`**`file_parametrize`**(*cls*, *directory*, *suffix*, *filter=None*, *wrapper=None*)

Creates a test for each file in a given directory. Call with any class that subclasses unittest.TestCase and which has a method called ‘‘ parametric_test‘‘, eg:

```
class MyTest(unittest.TestCase):
    def parametric_test(self,filename):
        self.assertTrue(os.path.exists(filename))

from ferenda.testutil import file_parametrize

file_parametrize(Parse,"test/files/legaluri",".txt")
```

For each .txt file in the directory `test/files/legaluri`, a corresponding test is created, which calls `parametric_test` with the full path to the .txt file as parameter.

Parameters

- **cls** (*class*) – TestCase to add the parametrized test to.
- **directory** (*str*) – The path to the files to turn into tests
- **suffix** – Suffix of the files that should be turned into tests (other files in the directory are ignored)

- **filter** – Will be called with the name of each matching file. If the `filter` callable returns True, no test is created
- **wrapper** – A unittest decorator like `unittest.skip()` or `unittest.expectedFailure()`.
- **wrapper** – callable (decorator)

`ferenda.testutil.`**`parametrize_repotester`**(*cls*)
Helper function to activate a `ferenda.testutil.RepoTester` based class (see the documentation for that class).

`ferenda.testutil.`**`testparser`**(*testcase*, *parser*, *filename*)
Helper function to test `FSMParser` based parsers.

16.3 Decorators

16.3.1 Decorators

Most of these decorators are intended to handle various aspects of a complete `parse()` implementation. Normally you should only use the `managedparsing()` decorator (if you even override the basic implementation). If you create separate actions aside from the standards (`download`, `parse`, `generate` et al), you should also use `action()` so that manage.py will be able to call it.

`ferenda.decorators.`**`timed`**(*f*)
Automatically log a statement of how long the function call takes

`ferenda.decorators.`**`recordlastdownload`**(*f*)
Automatically stores current time in `self.config.lastdownloaded`

`ferenda.decorators.`**`parseifneeded`**(*f*)
Makes sure the parse function is only called if needed, i.e. if the outfile is nonexistent or older than the infile(s), or if the user has specified in the config file or on the command line that it should be re-generated.

`ferenda.decorators.`**`render`**(*f*)
Handles the serialization of the `Document` object to XHTML+RDFa and RDF/XML files. Must be used in conjunction with `makedocument()`.

`ferenda.decorators.`**`handleerror`**(*f*)
Make sure any errors in `ferenda.DocumentRepository.parse()` are handled appropriately and do not stop the parsing of all documents.

`ferenda.decorators.`**`makedocument`**(*f*)
Changes the signature of the parse method to expect a Document object instead of a basefile string, and creates the object.

`ferenda.decorators.`**`managedparsing`**(*f*)
Use all standard decorators for parse() in the correct order (`makedocument()`, `parseifneeded()`, `timed()`, `render()`)

`ferenda.decorators.`**`action`**(*f*)
Decorator that marks a class or instance method as runnable by `ferenda.manager.run()`

`ferenda.decorators.`**`downloadmax`**(*f*)
Makes any generator respect the `downloadmax` config parameter.

`ferenda.decorators.`**`newstate`**(*state*)

16.4 Errors

16.4.1 Errors

These are the exceptions thrown by Ferenda. Any of the python built-in exceptions may be thrown as well, but exceptions in used third-party libraries should be wrapped in one of these.

exception ferenda.errors.**ParseError**
Raised when parse() fails in any way.

exception ferenda.errors.**FSMStateError**
Raised whenever the current state and the current symbol in a FSMParser configuration does not have a defined transition.

exception ferenda.errors.**DocumentRemovedError**
Raised whenever a particular document has been found to be removed – this can happen either during download() or parse() (which may be the case if there exists a physical document, but whose contents are essentially a placeholder saying that the document has been removed).

You can set the attribute dummyfile on this exception when raising it, preferably to the parsed_path that would be created, if not this exception had occurred.. If present, ferenda-build.py (or rather ferenda.manager.run()) will use this to create a dummy file at the indicated path. This prevents endless re-parsing of expired documents.

exception ferenda.errors.**PatchError**
Raised if a patch cannot be applied by patch_if_needed().

exception ferenda.errors.**NoDownloadedFileError**
Raised on an attempt to parse a basefile for which there doesn't exist a downloaded file.

exception ferenda.errors.**AttachmentNameError**
Raised whenever an invalid attachment name is used with any method of DocumentStore.

exception ferenda.errors.**AttachmentPolicyError**
Raised on any attempt to store an attachment using DocumentStore when storage_policy is not set to dir.

exception ferenda.errors.**ArchivingError**
Raised whenever an attempt to archive a document version using archive() fails (for example, because the archive version already exists).

exception ferenda.errors.**ValidationError**
Raised whenever a created document doesn't validate using the appropriate schema.

exception ferenda.errors.**TransformError**
Raised whenever a XSLT transformation fails for any reason.

exception ferenda.errors.**ExternalCommandError**
Raised whenever any invocation of an external commmand fails for any reason (including if the command line program doesn't exist).

exception ferenda.errors.**ConfigurationError**
Raised when a configuration file cannot be found in it's expected location or when it cannot be used due to corruption, file permissions or other reasons

exception ferenda.errors.**TriplestoreError**
Raised whenever communications with the triple store fails, for whatever reason.

exception ferenda.errors.**SparqlError**
Raised whenever a SPARQL query fails. The Exception should contain whatever error message that the Triple store returned, so the exact formatting may be dependent on which store is used.

exception ferenda.errors.**IndexingError**
Raised whenever an attempt to put text into the fulltext index fails.

exception ferenda.errors.**SearchingError**
Raised whenever an attempt to do a full-text search fails.

exception ferenda.errors.**SchemaConflictError**
Raised whenever a fulltext index is opened with repo arguments that result in a different schema than what's currently in use. Workaround this by removing the fulltext index and recreating.

exception ferenda.errors.**SchemaMappingError**
Raised whenever a given field in a schema cannot be mapped to or from the underlying native field object in an actual fulltextindex store.

16.5 Document repositories

16.5.1 ferenda.sources.general.Static – generate documents from your own .rst files

class ferenda.sources.general.**Static**(***kwargs*)
Generates documents from your own `.rst` files

The primary purpose of this docrepo is to provide a small set of static pages for a complete ferenda-based web site, like "About us", "Contact information", "Terms of service" or whatever else you need. The `download` step of this docrepo does not do anything, and it's `parse` step reads ReStructuredText (`.rst`) files from a local directory and converts them into XHTML+RDFa. From that point on, it works just like any other docrepo.

After enabling this, you should set the configuration parameter `staticdir` to the path of a directory where you keep your `.rst` files:

```
[static]
class = ferenda.sources.general.Static
staticdir = /var/www/mysite/static/rst
```

Note: If this configuration parameter is not set, this docrepo will use a small set of generic static pages, stored under `ferenda/res/static-pages` in the distribution. To get started, you can just copy this directory and set `staticdir` to point at your copy.

Every file present in `staticdir` results in a link in the site footer. The link text will be the title of the document, i.e. the first header in the `.rst` file.

16.5.2 ferenda.sources.general.Keyword – generate documents for keywords used by document in other docrepos

class ferenda.sources.general.**Keyword**(***kwargs*)
Implements support for 'keyword hubs', conceptual resources which themselves aren't related to any document, but to which other documents are related. As an example, if a docrepo has documents that each contains a set of keywords, and the docrepo parse implementation extracts these keywords as `dcterms:subject` resources, this docrepo creates a document resource for each of those keywords. The main content for the keyword may

come from the MediaWiki docrepo, and all other documents in any of the repos that refer to this concept resource are automatically listed.

16.5.3 ferenda.sources.general.MediaWiki – pull in commentary on documents and keywords from a MediaWiki instance

class ferenda.sources.general.**MediaWiki**(***kwargs*)
Downloads content from a Mediawiki system and converts it to annotations on other documents.

For efficient downloads, this docrepo requires that there exists a XML dump (created by dumpBackup.php) of the mediawiki contents that can be fetched over HTTP/HTTPS. Configure the location of this dump using the `mediawikiexport` parameter:

```
[mediawiki]
class = ferenda.sources.general.MediaWiki
mediawikiexport = http://localhost/wiki/allpages-dump.xml
```

16.5.4 ferenda.sources.general.Skeleton – generate skeleton documents for references from other documents

class ferenda.sources.general.**Skeleton**(***kwargs*)
Utility docrepo to fetch all RDF data from a triplestore (either our triple store, or a remote one, fetched through the combined ferenda atom feed), find out those resources that are referred to but not present in the data (usually older documents that are not available in electronic form), and create "skeleton entries" for those resources.

16.5.5 ferenda.sources.tech – repositories for technical standards

W3Standards

class ferenda.sources.tech.**W3Standards**(***kwargs*)

RFC

class ferenda.sources.tech.**RFC**(***kwargs*)

16.5.6 ferenda.sources.legal.eu – repositories for EU law

EurlexTreaties

class ferenda.sources.legal.eu.**EurlexTreaties**(***kwargs*)
Handles the foundation treaties of the European union.

EurlexCaselaw

class ferenda.sources.legal.eu.**EurlexTreaties**(***kwargs*)
Handles the foundation treaties of the European union.

16.5.7 ferenda.sources.legal.se – repositories for Swedish law

ARN

class ferenda.sources.legal.se.**ARN**(***kwargs*)
Hanterar referat från Allmänna Reklamationsnämnden, www.arn.se.

Modulen hanterar hämtande av referat från ARNs webbplats, omvandlande av dessa till XHTML1.1+RDFa, samt transformering till browserfärdig HTML5.

Direktiv

class ferenda.sources.legal.se.**Direktiv**(***kwargs*)
A composite repository containing DirTrips, DirAsp and DirRegeringen.

direktiv.DirTrips

class ferenda.sources.legal.se.direktiv.**DirTrips**(***kwargs*)
Downloads Direktiv in plain text format from http://rkrattsbaser.gov.se/dir/

direktiv.DirAsp

class ferenda.sources.legal.se.direktiv.**DirAsp**(***kwargs*)
Downloads Direktiv in PDF format from http://rkrattsdb.gov.se/kompdf/

direktiv.DirRegeringen

class ferenda.sources.legal.se.direktiv.**DirRegeringen**(***kwargs*)
Downloads Direktiv in PDF format from http://www.regeringen.se/

Ds

class ferenda.sources.legal.se.**Ds**(***kwargs*)

DV

class ferenda.sources.legal.se.**DV**(***kwargs*)

JK

class ferenda.sources.legal.se.**JK**(***kwargs*)

JO

class ferenda.sources.legal.se.**JO**(***kwargs*)
Hanterar beslut från Riksdagens Ombudsmän, www.jo.se

Modulen hanterar hämtande av beslut från JOs webbplats i PDF samt omvandlande av dessa till XHTML.

Kommitte

class ferenda.sources.legal.se.**Kommitte**(***kwargs*)

MyndFskr

class ferenda.sources.legal.se.**MyndFskr**(***kwargs*)

A abstract base class for fetching and parsing regulations from various swedish government agencies. These PDF documents often have a similar structure both graphically and linguistically, enabling us to parse them in a generalized way. (Downloading them often requires special-case code, though.)

myndfskr.SJVFS

class ferenda.sources.legal.se.myndfskr.**SJVFS**(***kwargs*)

myndfskr.DVFS

class ferenda.sources.legal.se.myndfskr.**DVFS**(***kwargs*)

myndfskr.FFFS

class ferenda.sources.legal.se.myndfskr.**FFFS**(***kwargs*)

myndfskr.ELSAKFS

class ferenda.sources.legal.se.myndfskr.**ELSAKFS**(***kwargs*)

myndfskr.NFS

class ferenda.sources.legal.se.myndfskr.**NFS**(***kwargs*)

myndfskr.STAFS

class ferenda.sources.legal.se.myndfskr.**STAFS**(***kwargs*)

myndfskr.SKVFS

class ferenda.sources.legal.se.myndfskr.**SKVFS**(***kwargs*)

Propositioner

class ferenda.sources.legal.se.**Propositioner**(***kwargs*)

propositioner.PropRegeringen

class ferenda.sources.legal.se.propositioner.**PropRegeringen**(***kwargs*)

propositioner.PropTrips

class ferenda.sources.legal.se.propositioner.**PropTrips**(***kwargs*)

propositioner.PropRiksdagen

class ferenda.sources.legal.se.propositioner.**PropRiksdagen**(***kwargs*)

SFS

class ferenda.sources.legal.se.**SFS**(***kwargs*)
Documentation to come.

A note about logging:

There are four additional loggers available ('paragraf', 'tabell', 'numlist' and 'rubrik'). By default, manager.py turns them off unless config.trace.[logname] is set. Do something like

./ferenda-build.py sfs parse 2009:924 –force –sfs-trace-rubrik

(sets the sfs.rubrik logger level to DEBUG) or

./ferenda-build.py sfs parse 2009:924 –force –sfs-trace-tabell=INFO

16.5.8 The Devel class

class ferenda.**Devel**(***kwargs*)
This module acts as a docrepo (and as such is easily callable from `ferenda-manager.py`), but instead of `download`, `parse`, `generate` et al, contains various tool commands that is useful for developing and debugging your own docrepo classes.

Use it by first enabling it:

```
./ferenda-build.py ferenda.Devel enable
```

And then run individual tools like:

```
./ferenda-build.py devel dumprdf path/to/xhtml/rdfa.xhtml
```

csvinventory(*alias*)
Create an inventory of documents, as a CSV file. Only documents that have been parsed and yielded some minimum amount of RDF metadata will be included.

Parameters **alias** (*str*) – Docrepo alias

dumprdf(*filename*, *format=u'turtle'*)
Extract all RDF data from a parsed file and dump it to stdout.

Parameters

- **filename** (*str*) – Full path of the parsed XHTML+RDFa file.

- **format** (*str*) – The serialization format for RDF data (same as for `rdflib.graph.Graph.serialize()`)

Example:

```
./ferenda-build.py devel dumprdf path/to/xhtml/rdfa.xhtml nt
```

dumpstore (*format=u'turtle'*)

Extract all RDF data from the system triplestore and dump it to stdout using the specified format.

Parameters **format** (*str*) – The serialization format for RDF data (same as for `ferenda.TripleStore.get_serialized()`).

Example:

```
./ferenda-build.py devel dumpstore nt > alltriples.nt
```

fsmparse (*functionname*, *source*)

Parse a list of text chunks using a named fsm parser and output the parse tree and final result to stdout.

Parameters

- **functionname** (*str*) – A function that returns a configured `FSMParser`
- **source** (*str*) – A file containing the text chunks, separated by double newlines

mkpatch (*alias*, *basefile*, *description*)

Create a patch file from downloaded or intermediate files. Before running this tool, you should hand-edit the intermediate file. If your docrepo doesn't use intermediate files, you should hand-edit the downloaded file instead. The tool will first stash away the intermediate (or downloaded) file, then re-run `parse()` (or `download_single()`) in order to get a new intermediate (or downloaded) file. It will then calculate the diff between these two versions and save it as a patch file in it's proper place (as determined by `config.patchdir`), where it will be picked up automatically by `patch_if_needed()`.

Parameters

- **alias** (*str*) – Docrepo alias
- **basefile** (*str*) – The basefile for the document to patch

Example:

```
./ferenda-build.py devel mkpatch myrepo basefile1 "Removed sensitive personal information"
```

parsestring (*string*, *citationpattern*, *uriformatter=None*)

Parse a string using a named citationpattern and print parse tree and optionally formatted uri(s) on stdout.

Parameters

- **string** (*str*) – The text to parse
- **citationpattern** (*str*) – The fully qualified name of a citationpattern
- **uriformatter** (*str*) – The fully qualified name of a uriformatter

---

Note: This is not implemented yet

Example:

```
./ferenda-build.py devel parsestring \
    "According to direktiv 2007/42/EU, ..." \
    ferenda.citationpatterns.eulaw
```

queryindex(*querystring*)
Query the system fulltext index and return the IDs/URIs for matching documents.

Parameters **querystring** (*str*) – The query

CHAPTER 17

Indices and tables

- *genindex*
- *modindex*
- *search*

f