regparser Documentation

Release beta

Author

Sep 27, 2017

Contents

Contents:

CHAPTER 1

Quick Start

Here's an example, using CFPB's regulation H.

```
git clone https://github.com/18F/regulations-parser.git
cd regulations-parser
pip install -r requirements.txt
eregs pipeline 12 1008 output_dir
```

At the end, you will have subdirectories `regulation`, `layer`, `diff`, and `notice` created under the directory named `output_dir`. These will mirror the JSON files sent to the API.

Quick Start with Modified Documents

Here's an example using FEC's regulation 110, showing how documents can be tweaked to pass the parser.

```
git clone https://github.com/18F/regulations-parser.git
cd regulations-parser
git clone https://github.com/micahsaul/fec_docs
pip install -r requirements.txt
echo "LOCAL_XML_PATHS = ['fec_docs']" >> local_settings.py
eregs pipeline 11 110 output_dir
```

If you review the history of the `fec_docs` repo, you'll see some of the types of changes that need to be made.

CHAPTER 2

Overview

Features

- Split regulation into paragraph-level chunks
- Create a tree which defines the hierarchical relationship between these chunks
- Layer for external citations – links to Acts, Public Law, etc.
- Layer for graphics – converting image references into federal register URLs
- Layer for internal citations – links between parts of this regulation
- Layer for interpretations – connecting regulation text to the interpretations associated with it
- Layer for key terms – pseudo headers for certain paragraphs
- Layer for meta info – custom data (some pulled from federal notices)
- Layer for paragraph markers – specifying where the initial paragraph marker begins and ends for each paragraph
- Layer for section-by-section analysis – associated analyses (from FR notices) with the text they are analyzing
- Layer for table of contents – a listing of headers
- Layer for terms – defined terms, including their scope
- Layer for additional formatting, including tables, “notes”, code blocks, and subscripts
- Build whole versions of the regulation from the changes found in final rules
- Create diffs between these versions of the regulations

Requirements

Python 2.7, 3.3, 3.4, 3.5. See `requirements.txt` and similar for specific library versions.

CHAPTER 3

Installation

Docker Install

For quick installation, consider installing from our Docker image. This image includes all of the relevant dependencies, wrapped up in a "container" for ease of installation. To run it, you'll need to have Docker installed, though the installation instructions for Linux, Mac, and Windows are relatively painless.

To run with Docker, there are some nasty configuration details which we'd like to hide behind a cleaner interface. Specifically, we want to provide a simple mechanism for collecting output, keep a cache around in between executions, allow input/output via stdio, and prevent containers from hanging around in between executions. To do that, we recommend creating a wrapper script and executing the parser through that wrapper.

For Linux and OS X, you could create a script, `eregs.sh`, that looks like:

```
#!/bin/sh
# Create a directory for the output
mkdir -p output
# Create a placeholder local_settings.py, if none exists
touch local_settings.py
# Execute docker with appropriate flags while passing in any arguments.
# --rm removes the container after execution
# -it makes the container interactive (particularly useful with --debug)
# -v mounts volumes for cache, output, and copies in the local settings
docker run --rm -it -v eregs-cache:/app/cache -v $PWD/output:/app/output -v $PWD/
↪local_settings.py:/app/code/local_settings.py eregs/parser $@
```

Remember to make that script executable:

```
chmod +x eregs.sh
```

To parse, run the wrapper script, `path/to/eregs.sh`, instead of `eregs` wherever instructed to in the rest of this documentation. Also, leave off the final argument in `pipeline` and `write_to` commands if you would like to see the results in the "output" directory.

From Source

Getting the Code and Development Libs

Download the source code from GitHub (e.g. `git clone [URL]`)

Make sure the `libxml` libraries are present. On Ubuntu/Debian, install it via

```
sudo apt-get install libxml2-dev libxslt-dev
```

Create a virtual environment (optional)

```
sudo pip install virtualenvwrapper
mkvirtualenv parser
```

Get the required libraries

```
cd regulations-parser
pip install -r requirements.txt
```

Run the parser

Using `pipeline`

```
eregs pipeline title part an/output/directory
```

or

```
eregs pipeline title part https://yourserver/
```

Example:

```
eregs pipeline 27 447 /output/path
```

Warning If using Docker and intending to write to the filesystem, remove the final parameter (`/output/path` above). All output will be written to the "/app/output" directory, which is mounted as "output" if you are using a script as described above.

`pipeline` pulls annual editions of regulations from the Government Printing Office and final rules from the Federal Register based on the part that you give it.

When you run `pipeline`, it:

1. Gets rules that exist for the regulation from the Federal Register API
2. Builds trees from annual editions of the regulation
3. Fills in any missing versions between annual versions by parsing final rules
4. Builds the layers for all these trees
5. Builds the diffs for all these trees, and

6. Writes the results to your output location

If the final parameter begins with `http://` or `https://`, output will be sent to that API. If it begins with `git://`, the output will be written as a git repository to that path. All other values will be treated as a file path; JSON files will be written in that directory. See output for more.

Settings

All of the settings listed in `regparser.web.settings.parser.py` can be overridden in a `local_settings.py` file. Current settings include:

- `META` - a dictionary of extra info which will be included in the "meta" layer. This is free-form, but could be used for copyright information, attributions, etc.
- `CFR_TITLES` - array of CFR Title names (used in the meta layer); not required as those provided are current
- `DEFAULT_IMAGE_URL` - string format used in the graphics layer; not required as the default should be adequate
- `IGNORE_DEFINITIONS_IN` - a dictionary mapping CFR part numbers to a list of terms that should *not* contain definitions. For example, if 'state' is a defined term, it may be useful to exclude the phrase 'shall state'. Terms associated with the constant, `ALL`, will be ignored in all CFR parts parsed.
- `INCLUDE_DEFINITIONS_IN` - a dictionary mapping CFR part numbers to a list of tuples containing (term, context) for terms that *are definitely definitions*. For example, a term that is succeeded by subparagraphs that define it rather than phraseology like "is defined as". Terms associated with the constant, `ALL`, will be included in all CFR parts parsed.
- `OVERRIDES_SOURCES` - a list of python modules (represented via string) which should be consulted when determining image urls. Useful if the Federal Register versions aren't pretty. Defaults to a `regcontent` module.
- `MACRO_SOURCES` - a list of python modules (represented via strings) which should be consulted if replacing chunks of XML in notices. This is more or less deprecated by `LOCAL_XML_PATHS`. Defaults to a `regcontent` module.
- `LOCAL_XML_PATHS` - a list of paths to search for notices from the Federal Register. This directory should match the folder structure of the Federal Register. If a notice is present in one of the local paths, that file will be used instead of retrieving the file, allowing for local edits, etc. to help the parser.

CHAPTER 4

Concepts

- **Diff**: a structure representing the changes between two regulation trees, describing which nodes were modified, deleted, or added.
- **Layer**: a grouping of extra information about the regulation, generally tied to specific text. For example, citations are a layer which refers to the text in a specific paragraph. There are also layers which apply to the entire tree, for example, the regulation letter. These are more or less a catch all for information which doesn't directly fit in the tree.
- **Rule**: a representation of the same concept as issued by the Federal Register. Sometimes called a **notice**. Rules change regulations, and have a great deal of meta data. Rules contain the contents, effective dates, and the authors of those changes. They can also potentially contain detailed analyses of each of the sections that changed.
- **Tree**: a representation of the regulation content. It's a recursive structure, where each component (part, subpart, section, paragraph, sub-sub-sub paragraph, etc.) is also a tree

CHAPTER 5

Command Line Usage

Assuming you have installed `regparser` via `pip` (either directly or indirectly via the requirements file), you should have access to the `eregs` program from the command line.

This interface is a wrapper around our various subcommands. For a list of all available commands, simply execute `eregs` without any parameters. This will also provide a brief description of the subcommand's purpose. To learn more, about each command's usage, run:

```
eregs <subcommand> --help
```

The Shared Index

Most of the subcommands make use of a shared index, or database, of partial computations. For example, rather than downloading and transforming XML files representing annual editions of a regulation with each run, the computation will be performed once and then stored within the index. All of these files are stored in the `.eregs_index` directory and can be safely deleted.

Further, these partial computations can depend on each other in the sense that one may be an essential input into another. When an "earlier" file (i.e. a dependency) is updated, it invalidates all of the partial computations which depended on it, which must now be re-built. The `eregs` command has logic to resolve missing or out-of-date dependencies automatically, by executing the appropriate subcommand which will update the necessary files.

The shared index allows computations to be built incrementally, as new data (e.g. a new final rule or annual edition) does not force all other versions of the regulation to be rebuilt. Moreover, by using this sort of shared database, we make no direct dependencies between commands. The command to generate "layer" data need not be aware if the depending regulation trees were generated from annual editions of the regulation, final rules, or something else.

The major caveat to this approach is that, if you are looking to change how the parser works, you will likely want it to re-compute specific data rather than relying on previous runs. This means you will need to `clear` the appropriate data to trigger rebuilds.

Shared Index Data

Here we document some of the file types within the shared index, so you know what needs to be cleared when editing the parser.

- `annual` - Transformed XML corresponding to the annual edition of regulations. This might need to be cleared if working on the XML transforms in `regparser.notice.preprocessors`
- `diff` - Structures representing Diffs between regulation trees. This most likely needs to be cleared if working on diff-computing code (`regparser.diff`)
- `layer` - These represent Layer data, with one file per regulation + version + layer type combination. These can be surgically removed depending on which `regparser.layer` has been edited
- `notice_xml` - Transformed XML corresponding to notices/final rules. These may need to be removed if working on the XML transforms in `regparser.notice.preprocessors`
- `rule_changes` - These structures are derived from the final rules in `notice_xml` and represent the set of amendments made for a regulation in that notice. These might need to be cleared if modifying any of the tree-building code (`regparser.tree`) or any amendment processing functions (in `regparser.notice`)
- `sxs` - A specific data representation for section-by-section analyses. These might need to be removed if modifying how SxS or notices more broadly are built (`regparser.notice`)
- `tree` - These represent the (whole) regulation at each version. Edits to tree-building code (notably `regparser.tree`) should lead you to remove these files.
- `version` - Each file here represents the dates and version identifier associated with each version of a regulation. These may need to be removed if working on the code which determines the order of regulation versions, delays between versions, etc. (mostly in `regparser.notice`)

Pipeline and its Components

The primary interface to the parser is the `pipeline` command, which pulls down all of the information needed to process a single regulation, parses it, and outputs the result in the requested format. The `pipeline` command gets its name from its operation – it effectively pulls in data and sends it through a "pipeline" of other commands, executing each in sequence. Each of these other commands can be executed independently, particularly useful if you are modifying the parser's workings.

- `versions` - Pull down and process a list of "versions" for a regulation, i.e. identifiers for when the regulation changed over time. This is a critical step as almost every other command uses this list of versions as a starting point for determining what work needs to be done. Each version has a specific identifier (referred to as the `version_id` or `document_number`) and effective date. These versions are generally associated with a Final Rule from the Federal Register. The process takes into account modifications to the effective dates by later rules. Output is in the index's `version` directory.
- `annual_editions` - Regulations are published once a year (technically, in batches, with a quarter published every three months). This command pulls down those annual editions of the regulation and associates the parsed output with the most recent version id. If multiple versions are effective in a single year, the last will be used (mod details around quarters.) Output is in the index's `tree` directory.
- `fill_with_rules` - If multiple versions of a regulation are effective in a single year, or if the annual edition has not been published yet, the parser will attempt to derive the changes from the Final Rules. Though fraught with error, this process is attempted for any versions which do not have an associated annual edition. The term "fill" comes from "filling" the gaps in the history of the regulation tree. Output is in the index's `tree` directory.
- `layers` - Now that the regulation's core content has been parsed, attempt to derive "layers" of additional data, such as internal citations, definitions, etc. Output is in the index's `layer` directory.

- `diffs` - The completed trees also allow the parser to compute the differences between trees. These data structures are created with this command, which saves its output in the index's `diff` directory.
- `write_to` - Once everything has been processed, we will want to send our results somewhere. If the final parameter begins with `http://` or `https://`, the parser will send the results as JSON to an HTTP API. If the final parameter begins with `git://`, the results will be serialized into a `git` repository and saved to the provided location. All other values are interpreted as a directory on disk; the output will be serialized to disk as JSON.

Many of the above commands depend on more fundamental commands, particularly commands to pull down and preprocess XML from the Federal Register and GPO. These commands are automatically called to fulfill dependencies generated by the above commands, but can also be executed separately. This is particularly useful if you need to re-import modified data.

- `preprocess_notice` - Given a final rule's document number, find the relevant XML (on disk or from the Federal Register), run it through a few preprocessing steps and save the results into the index's `notice_xml` directory.
- `fetch_annual_edition` - Given identifiers for which regulation and year, pull down the relevant XML, run it through the same preprocessing steps, and store the result into the index's `annual` directory.
- `parse_rule_changes` - Given a final rule's document number, convert the relevant XML file into a representation of the amendments, i.e. the instructions describing how the regulations is changing. Output stored in the index's `rule_changes` directory.
- `fetch_sxs` - Find and parse the "Section-by-Section Analyses" which are present in final rule associated with the provided document number. These are used to generate the SxS layer. Results stored in the index's `sxs` directory.

Tools

- `clear` - Removes content from the index. Useful if you have tweaked the parser's workings. Additional parameters can describe specific directories you would like to remove.
- `compare_to` - This command compares a set of local JSON files with a known copy, as stored in an instance of `regulations-core` (the API). The command will compare the requested JSON files and provide an interface for seeing the differences, if present.

CHAPTER 6

Developer Tasks

Building the documentation

For most tweaks, you will simply need to run the Sphinx documentation builder again.

```
pip install Sphinx
cd docs
make dirhtml
```

The output will be in `docs/_build/dirhtml`.

If you are adding new modules, you may need to re-run the skeleton build script first:

```
pip install Sphinx
sphinx-apidoc -F -o docs regparser/
```

Running Tests

As the parser is a complex beast, it has several hundred unit tests to help catch regressions. To run those tests, make sure you have first added all of the development requirements:

```
pip install -r requirements_dev.txt
```

Then, run py.test on all of the available unit tests:

```
py.test
```

If you'd like a report of test coverage, use the pytest-cov plugin:

```
py.test --cov-report term-missing --cov regparser
```

Note also that this library is continuously tested via Travis. Pull requests should rarely be merged unless Travis gives the green light.

CHAPTER 7

Additional Details

Here, we dive a bit deeper into some of the topics around the parser, so that you may use it in a production setting. We apologize in advance for somewhat out-of-date documentation.

Parsing Workflow

The parser first reads the file passed to it as a parameter and attempts to parse that into a structured tree of subparts, sections, paragraphs, etc. Following this, it will make a call to the Federal Register's API, retrieving a list of final rules (i.e. changes) that apply to this regulation. It then writes/saves parsed versions of those notices.

If this all worked well, we save the the parsed regulation and then generate and save all of the layers associated with its version. We then generate additional whole regulation trees and their associated layers for each final rule (i.e. each alteration to the regulation).

At the very end, we take all versions of the regulation we've built and compare each pair (both going forwards and backwards). These diffs are generated and then written to the API/filesystem/Git.

Output

The parser has three options for what it does with the parsed documents it creates, depending on the protocol it's give in `write_to`/`pipeline`, etc.

When no protocol is given (or the `file://` protocol is used), all of the created objects will be pretty-printed as JSON files and stored in subfolders of the provided path. Spitting out JSON files this way is a good way to track how tweaks to the parser might have unexpected effects on the output – just diff two such directories.

If the protocol is `http://` or `https://`, the output will be written to an API (running `regulations-core`) rather than the file system. The same JSON files are sent to the API as in the above method. This would be the method used once you are comfortable with the results (by testing the filesystem output).

A final method, a bit divergent from the other two, is to write the results as a git repository. To try this, use the `git://` protocol, telling the parser to write the versions of the regulation (*only*; layers, notices, etc. are not written)

as a git history. Each node in the parse tree will be written as a markdown file, with hierarchical information encoded in directories. This is an experimental feature, but has a great deal of potential.

Modifying Data

Our sources of data, through human and technical error, often contain problems for our parser. Over the parser's development, we've created several not-always-exclusive solutions. We have found that, in most cases, the easiest fix is to download and edit a *local* version of the problematic XML. Only if there's some complication in that method should you progress to the more complex strategies.

All of the paths listed in `LOCAL_XML_PATHS` are checked when fetching regulation notices. The file/directory names in these folders should mirror those found on federalregister.gov, (e.g. `articles/xml/201/131/725.xml`). Any changes you make to these documents (such as correcting XML tags, rewording amendment paragraphs, etc.) will be used as if they came from the Federal Register.

In addition, certain notices have *multiple* effective dates, meaning that different parts of the notice go into effect at different times. This complication is not handled automatically by the parser. Instead, you must manually copy the notice into two (or more) versions, such that 503.xml becomes 503-1.xml, 503-2.xml, etc. Each file must then be *manually* modified to change the effective date and remove sections that are not relevant to this date. We sometimes refer to this as "splitting" the notice.

Appendix Parsing

The most complicated segments of a regulation are their appendices, at least from a structural parsing perspective. This is because appendices are free-form, often with unique variations on sub-sections, headings, paragraph marker hierarchy, etc. Given all this, the parser does its best to determine *an* ordering and *a* hierarchy for the subsections/paragraphs contained within an appendix.

In general, if the parser can find a unique identifier or paragraph marker, it will note the paragraph/section accordingly. So "Part I: Blah Blah" becomes 1111-A-I, and "a. Some text" and "(a) Some text)" might become 1111-A-I-a. When the citable value of a paragraph cannot be determined (i.e. it has no paragraph marker), the paragraph will be assigned a number and prefaced with "p" (e.g. p1, p2). Similarly, headers become h1, h2, ...

This works out, but had numerous downsides. Most notably, as the citation for such paragraphs is arbitrary, determining changes to appendices is quite difficult (often requiring patches). Further, without guidance from paragraph markers/headers, the parser must make assumptions about the hierarchy of paragraphs. It currently uses some heuristics, such as headers indicating a new depth level, but is not always accurate.

Markdown/Plaintext-ifying

With some exceptions, we treat a plain-text version of the regulation as canon. By this, we mean that the *words* of the regulation count for much more than their presentation in the source documents. This allows us to build better tables of content, export data in more formats, and the other niceties associated with separating data from presentation.

At points, however, we need to encode non-plain text concepts into the plain-text regulation. These include displaying images, tables, offsetting blocks of text, and subscripting. To encode these concepts, we use a variation of Markdown.

Images become:

```
![Appendix A9](ER27DE11.000)
```

Tables become:

```
| Header 1 | Header 2|
---
| Cell 1, 1 | Cell 1, 2 |
```

Subscripts become:

```
P_{0}
```

etc.

Runtime

A quick note of warning: the parser was not optimized for speed. It performs many actions over and over, which can be **very** slow on very large regulations (such as CFPB's regulation Z). Further, regulations that have been amended a great deal cause further slow down, particularly when generating diffs (currently an n:super:*2* operation). Generally, parsing will take less than ten minutes, but in the extreme example of reg Z, it currently requires several hours.

Parsing Error Example

Let's say you are already in a good steady state, that you can parse the known versions of a regulation without problem. A new final rule is published in the federal register affecting your regulation. To make this concrete, we will use CFPB's regulation Z (12 CFR 1026), final rule 2014-18838.

The first step is to run the parser as we have before. We should configure it to send output to a local directory (see above). Once it runs, it will hit the federal register's API and should find the new notice. As described above, the parser first parses the file you give it, then it heads over to the federal register API, parses notices and rules found there, and then proceeds to compile additional versions of the regulation from them. So, as the parser is running (Z takes a long time), we can check its partial output. Notably, we can check the `notice/2014-18838` JSON file for accuracy.

In a browser, open https://www.federalregister.gov and search for the notice in question (you can do this by using the 2014-18838 identifier). Scroll through the page to find the list of changes – they will generally begin with "PART ..." and be offset from the rest of the text. In a text editor, look at the JSON file mentioned before.

The JSON file that describes our parsed notice has two relevant fields. The `amendments` field lists what *types* of changes are being made; it corresponds to AMDPAR tags (for reference). Looking at the web page, you should be able to map sentences like "Paragraph (b)(1)(ii)(A) and (B) are revised" to an appropriate PUT/POST/DELETE/etc. entry in the `amendments` field. If these do not match up, you know that there's an error parsing the AMDPARs. You will need to alter the XML for this notice to read how the parser can understand it. If the logic behind the change is too complicated, e.g. "remove the third semicolon and replace the fourth sentence", you will need to add a "patch" (see above).

In this case, the amendment parsing was correct, so we can continue to the second relevant field. The `changes` field includes the `content` of changes made (when adding or editing a paragraph). If all went well you should be able to relate all of the PUT/POST entries in the `amendments` section with an entry in the `changes` field, and the content of that entry should match the content from the federal register. Note that a single `amendment` may include multiple `changes` if the amendment is about a paragraph with children (sub-paragraphs).

Here we hit a problem, and have a few tip-offs. One of the entries in `amendments` was not present in the `changes` field. Further, one of the `changes` entries was something like "i. * * *". In addition, the "child_labels" of one of the entries doesn't make sense – it contains children which should not be contained. The parser must have skipped over some relevant information; we could try to deduce further but let's treat the parser as a black box and see if we can't

spot a problem in the web-hosted rule, first. You see, federalregister.gov uses XSLTs to take the raw XML (which we parse) to convert it into XHTML. If *we* have a problem, they might also.

We'll zero in on where we know our problem begins (based on the information investigating *changes*). We might notice that the text of the problem section is in italics, while those arround it (other sections which *do* parse correctly) are not. We might not. In any event, we need to look at the XML. On the federal register's site, there is a 'DEV' icon in the right sidebar and an 'XML' link in the modal. We're going to download this XML and put it where our parser knows to look (see the `LOCAL_XML_PATHS` setting). For example, if this setting is

```
LOCAL_XML_PATHS = ['fr-notices/']
```

we would need to save the XML file to `fr-notices/articles/xml/201/418/838.xml`, duplicating the directory structure found on the federal register. I recommend using a git repository and committing this "clean" version of the notice.

Now, edit the saved XML and jump to our problematic section. Does the XML structure here match sections we know work? It does not. Our "italic" tip off above was accurate. The problematic paragraphs are wrapped in `E` tags, which should not be present. Delete them and re-run the parser. You will see that this fixes our notice.

Generally, this will be the workflow. Something doesn't parse correctly and you must investigate. Most often, the problems will reside in unexpected XML structure. AMDPARs, which contain the list of changes may also need to be simplified. If the same type of change needs to be made for multiple documents, consider adding a corresponding rule to the parser – just test existing docs first.

Integration with regulations-core and regulations-site

TODO This section is rather out-of-date.

With the above examples, you should have been able to run the parser and generate some output. "But where's the website?" you ask. The parser was written to be as generic as possible, but integrating with `regulations-core` and `regulations-site` is likely where you'll want to end up. Here, we'll show one way to connect these applications up; see the individual repos for more configuration detail.

Let's set up `regulations-core` first. This is an API which will be used to both store and query the regulation data.

```
git clone https://github.com/18F/regulations-core.git
cd regulations-core
pip install -r requirements.txt  # pulls in python dependencies
./bin/django syncdb --migrate
./bin/django runserver 127.0.0.1:8888 &   # Starts the API
```

Then, we can configure the parser to write to this API and run it, here using the FEC example above

```
cd /path/to/regulations-parser
echo "API_BASE = 'http://localhost:8888/'" >> local_settings.py
eregs build_from fec_docs/1997CFR/CFR-1997-title11-vol1-part110.xml 11
```

Next up, we set up `regulations-site` to provide a webapp.

```
git clone https://github.com/18f/regulations-site.git
cd regulations-site
pip install -r requirements.txt
echo "API_BASE = 'http://127.0.0.1:8888/'" >> regulations/settings/local_settings.py
./run_server.sh
```

Then, navigate to http://localhost:8000/ in your browser to see the FEC reg.

CHAPTER 8

Parsing New Rules

Regulations are published, in full, annually; we rely on these annual editions to "synchronize" entire CFR parts. This works well when looking at the history of a regulation assuming that it has at most one change per year. When multiple final rules affect a single CFR part in a single year and when a *new* final rule has been issued, we don't have access to a canonical, entire regulation. To account for these situations, we have a parser for final rules, which attempts to figure out what section/paragraphs/etc. are changing and apply those changes to the previous version of the regulation to derive a new version.

Unfortunately, the changes are not encoded in a machine readable format, so the parser makes a best-effort, but tends to fall a bit short. In this document, we'll discuss what to expect from the parser and how to resolve common difficulties.

Fetching the Rule

Running the `pipeline` command will generally pull down and attempt to parse the relevant annual editions and final rules. It caches its results for a few days, so if a rule has only recently hit the Federal Register, you may need to run:

```
eregs clear
```

After running `pipeline`, you should see a version associated with the new rule in your output. If not, verify that the final rule is present on the Federal Register (our source of final rules). Looking in the right-hand column, you should find meta data associated with the final rule's publication date, effective date, entry type (must be "Rule"), and CFR references. If one of those fields is not present and you believe this to be in error, file a ticket on federalregister.gov's support page.

It's possible that running the `pipeline` causes an error. If you are familiar with Python, try running `eregs --debug pipeline` with the same parameters to get additional debugging output and to drop into a debugger at the point of error. Please file an issue and we will see if we can recreate the problem.

Viewing the Diff

Generally, eRegs will be able to create an appropriate version, but *won't* have found all of the appropriate changes. To make the verification process a bit easier, send the output to an instance of eRegs' UI. You can navigate to the "diff" view and compare the new rule to the previous version; the UI will highlight sections with changed text and tell you where *it* thinks changes have occurred. Open this view in conjunction with the text of the final rule and verify that the appropriate changes have been made.

We can also view more raw output representing the changes by investigating the output associated with `notices`. Run `pipeline` and send the results to a part of the file system, e.g.:

```
eregs pipeline 11 222 /tmp/eregs-output
```

and then inspect the `/tmp/eregs-output/notice` directory for a JSON file corresponding to the new rule. This data structure will contain keys associated with `amendments` (describing *how* the regulation is changing) and `changes` (describing the *content* of those changes).

Editing the Rule

Odds are that the parser did *not* pick up all of the changes present in the final rule. We can tweak the text of the rule to match align with the parser's expectations.

File Location

For initial edits, it'll make sense to modify the files directly within the index. These edits will trigger a rebuild on successive `pipeline` runs, but will be erased should the `clear` command ever be executed. To test out minor edits, modify the appropriate file in `.eregs_index/notice_xml`.

Once you would like to make those changes more permanent, we recommend you fork and checkout our shared notice-xml repository. Copy the final rule's XML (attainable via the "Dev" link from the Federal Register's UI) into a directory matching the structure.

For example, final rule 2014-18842 is represented by this XML: https://www.federalregister.gov/articles/xml/201/418/842.xml. To modify that, we'd save that XML file into `fr-notices/articles/xml/201/418/842.xml`.

We recommend committing this file in its original form to make it easy for future developers to understand what's changed. In any event, you'll need to inform the parser to look for your new file. To do so,

```
eregs clear   # remove the downloaded reference
echo 'LOCAL_XML_PATHS = ["path/to/fr-notices/"]' >> local_settings.py
```

Then re-run pipeline. This will alert the parser of the file's presence. You will only need to re-run the `pipeline` command on successive edits.

When all is said and done, we request you make a pull request to the shared `fr-notices` repository, which gets downloaded automatically by the parser.

Amendments

The complications around final rules arise largely from the amendment instructions (indicated by the `AMDPAR` tags in the XML). Unfortunately, we must attempt to parse these instructions, lest we will not know if paragraphs have been deleted, moved, etc. The `AMDParsing` logic attempts to find appropriate verbs ("revise", "correct", "add", "remove",

"reserve", "designate", etc.) and the paragraphs associated with those actions. So, the parser would understand an amendment like:

```
Section 1026.35 is amended by revising paragraph (b) introductory text,
adding new paragraph (b)(2), and removing paragraph (c).
```

In particular, it'd parse out as something like:

```
Context: 1026.35
Verb(PUT): amended, revising
Paragraph: 1026.35(b) introductory text
Verb(POST): adding
Paragraph: 1026.35(b)(2)
Verb(DELETE): removing
Paragraph: 1026.35(c)
```

We do not currently recognize concepts such as distinct sentences or specific words within a paragraph, so amendment instructions to "amend the fifth sentence" or "remove the last semicolon" cannot be understood. In these situations, it makes more sense to replace the text with something along the likes of "revise paragraph (b)" and include the entirety of the paragraph (rather than the single sentence, etc.).

We have also constructed two "artificial" amendment instructions to make this process easier.

- `[insert-in-order]` acts as a verb, indicating that the paragraph should be inserted in *textual* order (rather than by looking at the paragraph marker). This is particularly useful for modifications to definitions (which often do not contain paragraph markers).
- `[label:111-22-c]` acts as a very well defined paragraph. We can specifically target *any* paragraph this way for modification. Certain paragraphs are best defined by a specific keyterm or definition associated with them (rather than a paragraph marker). In these scenarios, we have a special syntax: `[label:111-22-keyterm(Special Term Here)]`

CHAPTER 9

Extension Points

The parser has several available extension points, with more added as the need arises. Take a look at our outline of the process for more information about the plugin system in general. Here we document specific extension points an example uses.

eregs_ns.parser.layers (deprecated)

List of strings referencing layer classes (generally implementing the abstract base class `regparser.layer.layer:Layer`).

Examples:

- ATF

This has been deprecated in favor of layers applicable to specific document types (see below).

eregs_ns.parser.layer.cfr

Layer classes (implementing the abstract base class `regparser.layer.layer:Layer`) which should apply the CFR documents.

eregs_ns.parser.layer.preamble

Layer classes (implementing the abstract base class `regparser.layer.layer:Layer`) which should apply the "preamble" documents (i.e. proposed rules).

eregs_ns.parser.preprocessors

List of strings referencing preprocessing classes (generally implementing the abstract base class `regparser.tree.xml_parser.preprocessors:PreProcessorBase`).

Examples:

- ATF
- FEC

Preprocessors may have a `plugin_order` attribute, an integer which defines the order in which the plugins are executed. Defaults to zero. Sorts ascending.

eregs_ns.parser.term_definitions

`dict: string->[(string,string)]`: List of phrases which *should* trigger a definition. Pair is of the form (term, context), where "context" refers to a substring match for a specific paragraph. e.g. ("bob", "text noting that it defines bob").

Examples:

- ATF
- EPA
- FEC

eregs_ns.parser.term_ignores

`dict: string->[string]`: List of phrases which shouldn't contain defined terms. Keyed by CFR part or `ALL`.

Examples:

- ATF
- FEC

eregs_ns.parser.test_suite

Extra modules to test with the `eregs full_tests` command.

Examples:

- ATF
- FEC

CHAPTER 10

regparser package

Subpackages

regparser.commands package

Submodules

regparser.commands.annual_editions module

regparser.commands.citations module

regparser.commands.clear module

regparser.commands.compare_to module

regparser.commands.compare_to.**compare**(*local_path*, *remote_url*, *prompt=True*)
Downloads and compares a local JSON file with a remote one. If there is a difference, notifies the user and prompts them if they want to see the diff

regparser.commands.compare_to.**file_to_json**(*path*)

regparser.commands.compare_to.**local_and_remote_generator**(*api_base*, *paths*)
Find all local files in *paths* and pair them with the appropriate remote file (prefixing with api_base). As the local files could be at any position in the file system, we back out directories until we hit one of the four root resource types (diff, layer, notice, regulation)

regparser.commands.compare_to.**path_to_json**(*path*)

regparser.commands.compare_to.**url_to_json**(*path*)

regparser.commands.current_version module

regparser.commands.dependency_resolver module

class regparser.commands.dependency_resolver.**DependencyResolver**(*dependency_path*)
Bases: object

Base class for objects which know how to "fix" missing dependencies.

PATH_PARTS = ()

has_resolution()

resolution()
This will generally call a command in an effort to resolve a dependency

regparser.commands.diffs module

regparser.commands.fetch_annual_edition module

regparser.commands.fetch_sxs module

regparser.commands.fill_with_rules module

regparser.commands.layers module

regparser.commands.parse_rule_changes module

regparser.commands.pipeline module

regparser.commands.preprocess_notice module

regparser.commands.sync_xml module

regparser.commands.versions module

regparser.commands.write_to module

Module contents

regparser.diff package

Submodules

regparser.diff.text module

regparser.diff.tree module

Module contents

regparser.grammar package

Submodules

regparser.grammar.amdpar module

regparser.grammar.amdpar.**generate_verb**(*word_list*, *verb*, *active*)
Short hand for making tokens.Verb from a list of trigger words

regparser.grammar.amdpar.**make_multiple**(*to_repeat*)
Shorthand for handling repeated tokens ('and', ',', 'through')

regparser.grammar.amdpar.**make_par_list**(*listify*, *force_text_field=False*)
Shorthand for turning a pyparsing match into a tokens.Paragraph

regparser.grammar.amdpar.**tokenize_override_ps**(*match*)
Create token.Paragraphs for the given override match

regparser.grammar.appendix module

regparser.grammar.appendix.**decimalize**(*characters*, *name*)

regparser.grammar.appendix.**parenthesize**(*characters*, *name*)

regparser.grammar.atomic module

Atomic components; probably shouldn't use these directly

regparser.grammar.delays module

class regparser.grammar.delays.**Delayed**
Bases: object

Placeholder token

class regparser.grammar.delays.**EffectiveDate**
Bases: object

Placeholder token

regparser.grammar.interpretation_headers module

regparser.grammar.terms module

regparser.grammar.tokens module

Set of Tokens to be used when parsing. @label is a list describing the depth of a paragraph/context. It follows: [Part, Subpart/Appendix/Interpretations, Section, p-level-1, p-level-2, p-level-3, p-level4, p-level5]

class regparser.grammar.tokens.**AndToken**
Bases: *regparser.grammar.tokens.Token*

The word 'and' can help us determine if a Context token should be a Paragraph token. Note that 'and' might also trigger the creation of a TokenList, which takes precedent

class regparser.grammar.tokens.**Context**(*label*, *certain=False*)
Bases: *regparser.grammar.tokens.Token*

Represents a bit of context for the paragraphs. This gets compressed with the paragraph tokens to define the full scope of a paragraph. To complicate matters, sometimes what looks like a Context is actually the entity which is being modified (i.e. a paragraph). If we are certain that this is only context, (e.g. "In Subpart A"), use 'certain'

certain

label

class regparser.grammar.tokens.**Paragraph**(*label=NOTHING*, *field=None*)
Bases: *regparser.grammar.tokens.Token*

Represents an entity which is being modified by the amendment. Label is a way to locate this paragraph (though see the above note). We might be modifying a field of a paragraph (e.g. intro text only, or title only;) if so, set the *field* parameter.

HEADING_FIELD = 'title'

KEYTERM_FIELD = 'heading'

TEXT_FIELD = 'text'

field

label

label_text()
Converts self.label into a string

classmethod make(*label=None*, *field=None*, *part=None*, *sub=None*, *section=None*, *paragraphs=None*, *paragraph=None*, *subpart=None*, *is_interp=None*, *appendix=None*)
label and field are the only "materialized" fields. Everything other field becomes part of the label, offering a more legible API. Particularly useful for writing tests

class regparser.grammar.tokens.**Token**
Bases: object

Base class for all tokens. Provides methods for pattern matching and copying this token

match(**types*, ***fields*)
Pattern match. self must be one of the types provided (if they were provided) and all of the fields must match (if fields were provided). If a successful match, returns self

class regparser.grammar.tokens.**TokenList**(*tokens*)
Bases: *regparser.grammar.tokens.Token*

Represents a sequence of other tokens, e.g. comma separated of created via "through"

tokens

class regparser.grammar.tokens.**Verb**(*verb*, *active*, *and_prefix=False*)
Bases: *regparser.grammar.tokens.Token*

Represents what action is taking place to the paragraphs

DELETE = 'DELETE'

DESIGNATE = 'DESIGNATE'

INSERT = 'INSERT'

KEEP = 'KEEP'

MOVE = 'MOVE'

POST = 'POST'

PUT = 'PUT'

RESERVE = 'RESERVE'

active

and_prefix

verb

regparser.grammar.tokens.**uncertain_label**(*label_parts*)
Convert a list of strings/Nones to a '-'-separated string with question markers to replace the Nones. We use this format to indicate uncertainty

regparser.grammar.unified module

Some common combinations

regparser.grammar.unified.**appendix_section**(*match*)
Appendices may have parenthetical paragraphs in its section number.

regparser.grammar.unified.**make_multiple**(*head*, *tail=None*, *wrap_tail=False*)
We have a recurring need to parse citations which have a string of terms, e.g. section 11(a), (b)(4), and (5). This function is a shorthand for setting these elements up

regparser.grammar.utils module

class regparser.grammar.utils.**DocLiteral**(*literal*, *ascii_text*)
Bases: pyparsing.Literal

Setting an objects name to a unicode string causes Sphinx to freak out. Instead, we'll replace with the provided (ascii) text.

regparser.grammar.utils.**Marker**(*txt*)

class regparser.grammar.utils.**Position**(*start*, *end*)
Bases: tuple

end
Alias for field number 1

start
Alias for field number 0

class regparser.grammar.utils.**QuickSearchable**(*expr*, *force_regex_str=None*)
Bases: pyparsing.ParseElementEnhance

Pyparsing's *scanString* (i.e. searching for a grammar over a string) tests each index within its search string. While that offers maximum flexibility, it is rather slow for our needs. This enhanced grammar type wraps other grammars, deriving from them a first regular expression to use when 'scanString'ing. This cuts search time considerably.

classmethod and_case(**first_classes*)
"And" grammars are relatively common; while we generally just want to look at their first terms, this decorator lets us describe special cases based on the class type of the first component of the clause

classmethod case(**match_classes*)
Add a "case" which will match grammars based on the provided class types. If there's a match, we'll execute the function

cases = [<function wordstart>, <function optional>, <function empty>, <function match_and>, <function match_or>, <f

classmethod initial_regex(*grammar*)
Given a Pyparsing grammar, derive a set of suitable initial regular expressions to aid our search. As grammars may *Or* together multiple sub-expressions, this always returns a *set* of possible regular expression

strings. This is _not_ a complete conversion to regexes nor does it account for every Pyparsing element; add as needed

scanString(*instring*, *maxMatches=None*, *overlap=False*)
Override *scanString* to attempt parsing only where there's a regex search match (as opposed to every index). Does not implement the full scanString interface.

regparser.grammar.utils.**SuffixMarker**(*txt*)

regparser.grammar.utils.**WordBoundaries**(*grammar*)

regparser.grammar.utils.**empty**(*grammar*)

regparser.grammar.utils.**has_re_string**(*grammar*)

regparser.grammar.utils.**keep_pos**(*expr*)
Transform a pyparsing grammar by inserting an attribute, "pos", on the match which describes position information

regparser.grammar.utils.**line_start**(*grammar*)

regparser.grammar.utils.**literal**(*grammar*)

regparser.grammar.utils.**match_and**(*grammar*)

regparser.grammar.utils.**match_or**(*grammar*)

regparser.grammar.utils.**optional**(*grammar*)

regparser.grammar.utils.**parse_position**(*source*, *location*, *tokens*)
A pyparsing parse action which pulls out (and removes) the position information and replaces it with a Position object

regparser.grammar.utils.**suppress**(*grammar*)

regparser.grammar.utils.**wordstart**(*grammar*)
Optimization: WordStart is generally followed by a more specific identifier. Rather than searching for the start of a word alone, search for that identifier as well

Module contents

regparser.history package

Submodules

regparser.history.annual module

regparser.history.delays module

class regparser.history.delays.**FRDelay**
Bases: *regparser.history.delays.FRDelay*

modifies_notice_xml(*notice_xml*)
Calculates whether the fr citation is within the provided NoticeXML

regparser.history.delays.**delays_in_sentence**(*sent*)
Tokenize the provided sentence and check if it is a format that indicates that some notices have changed. This format is: ... "effective date" ... FRNotices ... "delayed" ... (UntilDate)

regparser.history.notices module

regparser.history.versions module

class regparser.history.versions.**Version**
Bases: *regparser.history.versions.Version*

static from_json(*json_str*)

is_final

is_proposal

json()

static parents_of(*versions*)
A "parent" of a version is the version which it builds atop. Versions can only build on final versions. Assume the versions are already sorted

Module contents

regparser.index package

Submodules

regparser.index.dependency module

regparser.index.entry module

regparser.index.xml_sync module

Module contents

regparser.layer package

Submodules

regparser.layer.def_finders module

Parsers for finding a term that's being defined within a node

class regparser.layer.def_finders.**DefinitionKeyterm**(*parent*)
Bases: object

Matches definitions identified by being a first-level paragraph in a section with a specific title

find(*node*)

class regparser.layer.def_finders.**ExplicitIncludes**
Bases: *regparser.layer.def_finders.FinderBase*

Definitions can be explicitly included in the settings. For example, say that a paragraph doesn't indicate that a certain phrase is a definition; we can define INCLUDE_DEFINITIONS_IN in our settings file, which will be checked here.

find(*node*)

class regparser.layer.def_finders.**FinderBase**
Bases: object

Base class for all of the definition finder classes. Defines the interface they must implement

find(*node*)
Given a Node, pull out any definitions it may contain as a list of Refs

class regparser.layer.def_finders.**Ref**
Bases: *regparser.layer.def_finders.Ref*

A reference to a defined term. Keeps track of the term, where it was found and the term's position in that node's text

end

position

class regparser.layer.def_finders.**ScopeMatch**(*finder*)
Bases: *regparser.layer.def_finders.FinderBase*

We know these will be definitions because the scope of the definition is spelled out. E.g. 'for the purposes of XXX, the term YYY means'

find(*node*)

class regparser.layer.def_finders.**SmartQuotes**(*stack*)
Bases: *regparser.layer.def_finders.FinderBase*

Definitions indicated via smart quotes

find(*node*)

has_def_indicator()
With smart quotes, we catch some false positives, phrases in quotes that are not terms. This extra test lets us know that a parent of the node looks like it would contain definitions.

class regparser.layer.def_finders.**XMLTermMeans**(*existing_refs=None*)
Bases: *regparser.layer.def_finders.FinderBase*

Namespace for a matcher for e.g. '<E>XXX</E> means YYY'

find(*node*)

pos_start(*needle*, *haystack*)
Search for the first instance of *needle* in the *haystack* excluding any overlaps from *self.exclusions*. Implicitly returns None if it can't be found

regparser.layer.external_citations module

class regparser.layer.external_citations.**ExternalCitationParser**(*tree*, ***context*)
Bases: *regparser.layer.layer.Layer*

External Citations are references to documents outside of eRegs. See *external_types* for specific types of external citations

process(*node*)

shorthand = 'external-citations'

regparser.layer.external_types module

Parsers for various types of external citations. Consumed by the external citation layer

class regparser.layer.external_types.**CFRFinder**
Bases: *regparser.layer.external_types.FinderBase*

Code of Federal Regulations. Explicitly ignore any references within this part

CITE_TYPE = 'CFR'

find(*node*)

class regparser.layer.external_types.**Cite**(*cite_type*, *start*, *end*, *components*, *url*)
Bases: tuple

cite_type
Alias for field number 0

components
Alias for field number 3

end
Alias for field number 2

start
Alias for field number 1

url
Alias for field number 4

class regparser.layer.external_types.**CustomFinder**
Bases: *regparser.layer.external_types.FinderBase*

Explicitly configured citations; part of settings

CITE_TYPE = 'OTHER'

find(*node*)

class regparser.layer.external_types.**FDSYSFinder**
Bases: object

Common parent class to Finders which generate an FDSYS url based on matching a PyParsing grammar

CONST_PARAMS
Constant parameters we pass to the FDSYS url; a dict

GRAMMAR
A pyparsing grammar with relevant components labeled

find(*node*)

class regparser.layer.external_types.**FinderBase**
Bases: object

Base class for all of the external citation parsers. Defines the interface they must implement.

CITE_TYPE
A constant to represent the citations this produces.

find(*node*)
Give a Node, pull out any external citations it may contain as a generator of Cites

class regparser.layer.external_types.**PublicLawFinder**
Bases: *regparser.layer.external_types.FDSYSFinder*, *regparser.layer.external_types.FinderBase*

Public Law

CITE_TYPE = 'PUBLIC_LAW'

CONST_PARAMS = {'collection': 'plaw', 'lawtype': 'public'}

GRAMMAR = QuickSearchable:({{{{Suppress:({{WordStart 'Public'} WordEnd}) Suppress:({{WordStart 'Law'} WordEnd

class regparser.layer.external_types.**StatutesFinder**
Bases: *regparser.layer.external_types.FDSYSFinder*, *regparser.layer.external_types.FinderBase*

Statutes at large

CITE_TYPE = 'STATUTES_AT_LARGE'

CONST_PARAMS = {'collection': 'statute'}

GRAMMAR = QuickSearchable:({{W:(0123...) Suppress:("Stat.")} W:(0123...)})

class regparser.layer.external_types.**USCFinder**
Bases: *regparser.layer.external_types.FDSYSFinder*, *regparser.layer.external_types.FinderBase*

U.S. Code

CITE_TYPE = 'USC'

CONST_PARAMS = {'collection': 'uscode'}

GRAMMAR = QuickSearchable:({{{W:(0123...) "U.S.C."} Suppress:(["Chapter"])} W:(0123...)})

class regparser.layer.external_types.**UrlFinder**
Bases: *regparser.layer.external_types.FinderBase*

Any raw urls in the text

CITE_TYPE = 'OTHER'

PUNCTUATION = '.,;?\"')-'

REGEX = <_sre.SRE_Pattern object>

find(*node*)

regparser.layer.external_types.**fdsys_url**(***params*)
Generate a URL to an FDSYS redirect

regparser.layer.formatting module

Find and abstracts formatting information from the regulation tree. In many ways, this is like a markdown parser.

class regparser.layer.formatting.**Dashes**
Bases: *regparser.layer.formatting.PlaintextFormatData*

E.g. Some text some text_____

REGEX = <_sre.SRE_Pattern object>

match_data(*match*)

class regparser.layer.formatting.**FencedData**
Bases: *regparser.layer.formatting.PlaintextFormatData*

E.g. `note Line 1 Line 2 `

REGEX = <_sre.SRE_Pattern object>

match_data(*match*)

class regparser.layer.formatting.**Footnotes**
Bases: *regparser.layer.formatting.PlaintextFormatData*

E.g. [^4](Contents of footnote) The footnote may also contain parens if they are escaped with a backslash

REGEX = <_sre.SRE_Pattern object>

match_data(*match*)

class regparser.layer.formatting.**Formatting**(*tree*, ***context*)
Bases: *regparser.layer.layer.Layer*

Layer responsible for tables, subscripts, and other formatting-related information

process(*node*)

shorthand = 'formatting'

class regparser.layer.formatting.**HeaderStack**
Bases: *regparser.tree.priority_stack.PriorityStack*

Used to determine Table Headers – indeed, they are complicated enough to warrant their own stack

unwind()

class regparser.layer.formatting.**PlaintextFormatData**
Bases: object

Base class for formatting information which can be derived from the plaintext of a regulation node

REGEX
Regular expression used to find matches in the plain text

match_data(*match*)
Derive data structure (as a dict) from the regex match

process(*text*)
Find all matches of self.REGEX, transform them into the appropriate data structure, return these as a list

class regparser.layer.formatting.**Subscript**
Bases: *regparser.layer.formatting.PlaintextFormatData*

E.g. a_{0}

REGEX = <_sre.SRE_Pattern object>

match_data(*match*)

class regparser.layer.formatting.**Superscript**
Bases: *regparser.layer.formatting.PlaintextFormatData*

E.g. x^{2}

REGEX = <_sre.SRE_Pattern object>

match_data(*match*)

class regparser.layer.formatting.**TableHeaderNode**(*text*, *level*)
Bases: object

Represents a cell in a table's header

height()

width()

regparser.layer.formatting.**build_header**(*xml_nodes*)
Builds a TableHeaderNode tree, with an empty root. Each node in the tree includes its colspan/rowspan

regparser.layer.formatting.**build_header_rowspans**(*tree_root*, *max_height*)
The following table is an example of why we need a relatively complicated approach to setting rowspan:

|R1C1 |R1C2 | |R2C1|R2C2|R2C3 |R2C4 | | | |R3C1|R3C2|R3C3|R3C4|

If we set the rowspan of each node to:

```
max_height - node.height() - node.level + 1
```

R1C1 will end up with a rowspan of 2 instead of 1, because of difficulties handling the implicit rowspans for R2C1 and R2C2.

Instead, we generate a list of the paths to each leaf and then set rowspan based on that.

Rowspan for leaves is `max_height - node.height() - node.level + 1`, and for root is simply 1. Other nodes' rowspans are set to the level of the node after them minus their own level.

regparser.layer.formatting.**node_to_table_xml_els**(*node*)
Search in a few places for GPOTABLE xml elements

regparser.layer.formatting.**table_xml_to_data**(*xml_node*)
Construct a data structure of the table data. We provide a different structure than the native XML as the XML encodes too much logic. This structure can be used to generate semi-complex tables which could not be generated from the markdown above

regparser.layer.formatting.**table_xml_to_plaintext**(*xml_node*)
Markdown representation of a table. Note that this doesn't account for all the options needed to display the table properly, but works fine for simple tables. This gets included in the reg plain text

regparser.layer.graphics module

regparser.layer.internal_citations module

class regparser.layer.internal_citations.**InternalCitationParser**(*tree*, *cfr_title*, ***context*)
Bases: *regparser.layer.layer.Layer*

parse(*text*, *label*, *title=None*)
Parse the provided text, pulling out all the internal (self-referential) citations.

pre_process()
As a preprocessing step, run through the entire tree, collecting all labels.

process(*node*)

remove_missing_citations(*citations*, *text*)
Remove any citations to labels we have not seen before (i.e. those collected in the pre_processing stage)

shorthand = 'internal-citations'

static strip_whitespace(*text*, *citations*)
Modifies the offsets to exclude any trailing whitespace. Modifies the offsets in place.

regparser.layer.interpretations module

regparser.layer.key_terms module

class regparser.layer.key_terms.**KeyTerms**(*tree*, ***context*)
Bases: *regparser.layer.layer.Layer*

static is_definition(*node*, *keyterm*)
A definition might be masquerading as a keyterm. Do not allow this

classmethod keyterm_in_node(*node*, *ignore_definitions=True*)

process(*node*)
Get keyterms if we have text in the node that preserves the <E> tags.

shorthand = u'keyterms'

regparser.layer.key_terms.**keyterm_in_text**(*tagged_text*)
Pull out the key term of the provided markup using a regex. The XML <E> tags that indicate keyterms are also used for italics, which means some non-key term phrases would be lumped in. We eliminate them here.

regparser.layer.layer module

class regparser.layer.layer.**Layer**(*tree*, ***context*)
Bases: object

Base class for all of the Layer generators. Defines the interface they must implement

build(*cache=None*)

builder(*node*, *cache=None*)

static convert_to_search_replace(*matches*, *text*, *start_fn*, *end_fn*)
We'll often have a bunch of text matches based on offsets. To use the "search-replace" encoding (which is a bit more resilient to minor variations in text), we need to convert these offsets into "locations" – i.e. of all of the instances of a string in this text, which should be matched. Yields *SearchReplace* tuples

pre_process()
Take the whole tree and do any pre-processing

process(*node*)
Construct the element of the layer relevant to processing the given node, so it returns (pargraph_id, layer_content) or None if there is no relevant information.

shorthand
Unique identifier for this layer

class regparser.layer.layer.**SearchReplace**(*text*, *locations*, *representative*)
Bases: tuple

locations
Alias for field number 1

representative
Alias for field number 2

text
Alias for field number 0

regparser.layer.meta module

class regparser.layer.meta.**Meta**(*tree*, *cfr_title*, *version*, ***context*)
Bases: *regparser.layer.layer.Layer*

effective_date()

process(*node*)
If this is the root element, add some 'meta' information about this regulation, including its cfr title, effective date, and any configured info

shorthand = 'meta'

regparser.layer.model_forms_text module

regparser.layer.paragraph_markers module

class regparser.layer.paragraph_markers.**ParagraphMarkers**(*tree*, ***context*)
Bases: *regparser.layer.layer.Layer*

process(*node*)
Look for any leading paragraph markers.

shorthand = 'paragraph-markers'

regparser.layer.paragraph_markers.**marker_of**(*node*)
Try multiple potential marker formats. See if any apply to this node.

regparser.layer.scope_finder module

class regparser.layer.scope_finder.**ScopeFinder**
Bases: object

Useful for determining the scope of a term

add_subparts(*root*)
Document the relationship between sections and subparts

determine_scope(*stack*)

scope_of_text(*text*, *label_struct*, *verify_prefix=True*)
Given specific text, try to determine the definition scope it indicates. Implicit return None if none is found.

subpart_scope(*label_parts*)
Given a label, determine which sections fall under the same subpart

regparser.layer.section_by_section module

class regparser.layer.section_by_section.**SectionBySection**(*tree*, *notices*, ***context*)
Bases: *regparser.layer.layer.Layer*

process(*node*)
Determine which (if any) section-by-section analyses would apply to this node.

shorthand = 'analyses'

regparser.layer.table_of_contents module

class regparser.layer.table_of_contents.**TableOfContentsLayer**(*tree*, ***context*)
Bases: *regparser.layer.layer.Layer*

check_toc_candidacy(*node*)
To be eligible to contain a table of contents, all of a node's children must have a title element. If one of the children is an empty subpart, we check all it's children.

process(*node*)
Create a table of contents for this node, if it's eligible. We ignore subparts.

shorthand = 'toc'

regparser.layer.terms module

class regparser.layer.terms.**Inflected**(*singular*, *plural*)
Bases: tuple

plural
Alias for field number 1

singular
Alias for field number 0

class regparser.layer.terms.**ParentStack**
Bases: *regparser.tree.priority_stack.PriorityStack*

Used to keep track of the parents while processing nodes to find terms. This is needed as the definition may need to find its scope in parents.

parent_of(*node*)

unwind()
No collapsing needs to happen.

class regparser.layer.terms.**Terms**(**args*, ***kwargs*)
Bases: *regparser.layer.layer.Layer*

ENDS_WITH_WORDCHAR = <_sre.SRE_Pattern object>

STARTS_WITH_WORDCHAR = <_sre.SRE_Pattern object>

applicable_terms(*label*)
Find all terms that might be applicable to nodes with this label. Note that we don't have to deal with subparts as subpart_scope simply applies the definition to all sections in a subpart

calculate_offsets(*text*, *applicable_terms*, *exclusions=None*, *inclusions=None*)
Search for defined terms in this text, including singular and plural forms of these terms, with a preference for all larger (i.e. containing) terms.

excluded_offsets(*node*)
We explicitly exclude certain chunks of text (for example, words we are defining shouldn't have links appear within the defined term.) More will be added in the future

ignored_offsets(*cfr_part*, *text*)
Return a list of offsets corresponding to the presence of an "ignored" phrase in the text

inflected(*term*)
Check the memoized Inflected version of the provided term

is_exclusion(*term*, *node*)
Some definitions are exceptions/exclusions of a previously defined term. At the moment, we do not want to include these as they would replace previous (correct) definitions. We also remove terms which are inside an instance of the IGNORE_DEFINITIONS_IN setting

look_for_defs(*node*, *stack=None*)
Check a node and recursively check its children for terms which are being defined. Add these definitions to self.scoped_terms.

node_definitions(*node*, *stack=None*)
Find defined terms in this node's text.

pre_process()
Step through every node in the tree, finding definitions. Also keep track of which subpart we are in. Finally, document all defined terms.

process(*node*)
Determine which (if any) definitions would apply to this node, then find if any of those terms appear in this node

shorthand = u'terms'

Module contents

regparser.notice package

Submodules

regparser.notice.address module

regparser.notice.build module

regparser.notice.build_appendix module

regparser.notice.build_interp module

regparser.notice.changes module

This module contains functions to help parse the changes in a notice. Changes are the exact details of how the pargraphs, sections etc. in a regulation have changed.

class regparser.notice.changes.**Change**(*label_id*, *content*)
Bases: tuple

content
Alias for field number 1

label_id
Alias for field number 0

class regparser.notice.changes.**NoticeChanges**
Bases: object

Notice changes.

add_change(*amdpar_xml*, *change*)
Track another change. This is cognizant of the fact that a single label can have more than one change. Do not add the same change twice (as may occur if both the parent and child are marked as added)

regparser.notice.changes.**bad_label**(*node*)
Look through a node label, and return True if it's a badly formed label. We can do this because we know what type of character should up at what point in the label.

regparser.notice.changes.**create_add_amendment**(*amendment*, *subpart_label=None*)
An amendment comes in with a whole tree structure. We break apart the tree here (this is what flatten does), convert the Node objects to JSON representations. This ensures that each amendment only acts on one node. In addition, this futzes with the change's field when stars are present.

regparser.notice.changes.**create_field_amendment**(*label*, *amendment*)
If an amendment is changing just a field (text, title) then we don't need to package the rest of the paragraphs with it. Those get dealt with later, if appropriate.

regparser.notice.changes.**create_reserve_amendment**(*amendment*)
Create a RESERVE related amendment.

regparser.notice.changes.**create_subpart_amendment**(*subpart_node*)
Create an amendment that describes a subpart. In particular when the list of nodes added gets flattened, each node specifies which subpart it's part of.

regparser.notice.changes.**find_candidate**(*root*, *label_last*, *amended_labels*)
Look through the tree for a node that has the same paragraph marker as the one we're looking for (and also has no children). That might be a mis-parsed node. Because we're parsing partial sections in the notices, it's likely we might not be able to disambiguate between paragraph markers.

regparser.notice.changes.**find_misparsed_node**(*section_node*, *label*, *change*, *amended_labels*)
Nodes can get misparsed in the sense that we don't always know where they are in the tree or have their correct label. The first part corrects markerless labeled nodes by updating the node's label if the source text has been changed to include the markerless paragraph (ex. 123-44-p6 for paragraph 6). we know this because *label* here is parsed from that change. The second part uses label to find a candidate for a mis-parsed node and creates an appropriate change.

regparser.notice.changes.**find_subpart**(*amdpar_tag*)
Look amongst an amdpar tag's siblings to find a subpart.

regparser.notice.changes.**fix_section_node**(*paragraphs*, *amdpar_xml*)
When notices are corrected, the XML for notices doesn't follow the normal syntax. Namely, pargraphs aren't inside section tags. We fix that here, by finding the preceding section tag and appending paragraphs to it.

regparser.notice.changes.**flatten_tree**(*node_list*, *node*)
Flatten a tree, removing all hierarchical information, making a list out of all the nodes.

regparser.notice.changes.**format_node**(*node*, *amendment*, *parent_label=None*)
Format a node into a dict, and add in amendment information.

regparser.notice.changes.**impossible_label**(*n*, *amended_labels*)
Return True if n is not in the same family as amended_labels.

regparser.notice.changes.**match_labels_and_changes**(*amendments*, *section_node*)
Given the list of amendments, and the parsed section node, match the two so that we're only changing what's been flagged as changing. This helps eliminate paragraphs that are just stars for positioning, for example.

regparser.notice.changes.**new_subpart_added**(*amendment*)
Return True if label indicates that a new subpart was added

regparser.notice.changes.**node_to_dict**(*node*)
Convert a node to a dictionary representation. We skip the children, turning them instead into a list of labels instead.

regparser.notice.changes.**resolve_candidates**(*amend_map*, *warn=True*)
Ensure candidate isn't actually accounted for elsewhere, and fix it's label.

regparser.notice.compiler module

Notices indicate how a regulation has changed since the last version. This module contains code to compile a regulation from a notice's changes.

class regparser.notice.compiler.**RegulationTree**(*previous_tree*)
Bases: object

This encapsulates a regulation tree, and methods to change that tree.

static add_child(*children*, *node*, *order=None*)
Add a child to the children, and sort appropriately. This is used for non-root nodes.

add_node(*node*, *parent_label=None*)
Add an entirely new node to the regulation tree. Accounts for placeholders, reserved nodes,

add_to_root(*node*)
Add a child to the root of the tree.

contains(*label*)
Is this label already in the tree? label can be a list or a string

create_empty_node(*node_label*)
In rare cases, we need to flush out the tree by adding an empty node. Returns the created node

create_new_subpart(*subpart_label*)
Create a whole new subpart.

delete(*label_id*)
Delete the node with label_id from the tree.

delete_from_parent(*node*)
Delete node from it's parent, effectively removing it from the tree.

find_node(*label*)

get_parent(*node*)
Get the parent of a node. Returns None if parent not found.

insert_in_order(*node*)
Add a new node, but determine its position in its parent by looking at the siblings' texts

keep(*labels*)
The 'KEEP' verb tells us that a node should not be removed (generally because it would had we dropped the children of its parent). "Keeping" those nodes makes sure they do not disappear when editing their parent

move(*origin*, *destination*)
Move a node from one part in the tree to another.

move_to_subpart(*label*, *subpart_label*)
Move an existing node to another subpart. If the new subpart doesn't exist, create it.

replace_node_and_subtree(*node*)
Replace an existing node in the tree with node.

replace_node_heading(*label*, *change*)
A node's heading is it's keyterm. We handle this here, but not well, I think.

replace_node_text(*label*, *change*)
Replace just a node's text.

replace_node_title(*label*, *change*)
Replace just a node's title.

reserve(*label_id*, *node*)
Reserve either an existing node (by replacing it) or reserve by adding a new node. When a node is reserved, it's represented in the FR XML. We simply use that representation here instead of doing something else.

regparser.notice.compiler.**compile_regulation**(*previous_tree*, *notice_changes*)
Given a last full regulation tree, and the set of changes from the next final notice, construct the next full regulation tree.

regparser.notice.compiler.**dict_to_node**(*node_dict*)
Convert a dictionary representation of a node into a Node object if it contains the minimum required fields. Otherwise, pass it through unchanged.

regparser.notice.compiler.**get_parent_label**(*node*)
Given a node, get the label of it's parent.

regparser.notice.compiler.**is_interp_placeholder**(*node*)
Interpretations may have nodes that exist purely to enforce structure. Knowing if a node is such a placeholder makes it easier to know if a POST should really just modify the existing placeholder.

regparser.notice.compiler.**is_reserved_node**(*node*)
Return true if the node is reserved.

regparser.notice.compiler.**make_label_sortable**(*label*, *roman=False*)
Make labels sortable, but converting them as appropriate. For example, "45Ai33b" becomes (45, "A", "i", 33, "b"). Also, appendices have labels that look like 30(a), we make those appropriately sortable.

regparser.notice.compiler.**make_root_sortable**(*label*, *node_type*)
Child nodes of the root contain nodes of various types, these need to be sorted correctly. This returns a tuple to help sort these first level nodes.

regparser.notice.compiler.**node_text_equality**(*left*, *right*)
Do these two nodes have the same text fields? Accounts for Nones

regparser.notice.compiler.**one_change**(*reg*, *label*, *change*)
Notices are generally composed of many changes; this method handles a single change to the tree.

regparser.notice.compiler.**overwrite_marker**(*origin*, *new_label*)
The node passed in has a label, but we're going to give it a new one (new_label). This is necessary during node moves.

regparser.notice.compiler.**replace_first_sentence**(*text*, *replacement*)
Replace the first sentence in text with replacement. This makes some incredibly simplifying assumptions - so buyer beware.

regparser.notice.compiler.**replace_node_field**(*reg*, *label*, *change*)
Call one of the field appropriate methods if we're changing just a field on a node.

regparser.notice.compiler.**sort_labels**(*labels*)
Deal with higher up elements first.

regparser.notice.dates module

regparser.notice.dates.**fetch_dates**(*xml*)
Pull out any dates (and their types) from the XML. Not all notices have all types of dates, some notices have multiple dates of the same type.

regparser.notice.dates.**parse_date_sentence**(*sentence*)
Return the date type + date in this sentence (if one exists).

regparser.notice.diff module

regparser.notice.encoder module

class regparser.notice.encoder.**AmendmentEncoder**(*skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *encoding='utf-8'*, *default=None*)

Bases: json.encoder.JSONEncoder

Custom JSON encoder to handle Amendment objects

default(*obj*)

regparser.notice.fake module

regparser.notice.sxs module

regparser.notice.sxs.**add_spaces_to_title**(*title*)
Federal Register often seems to miss spaces in the title of SxS sections. Make sure spaces get added if appropriate

regparser.notice.sxs.**build_section_by_section**(*sxs*, *fr_start_page*, *previous_label*)
Given a list of xml nodes in the section by section analysis, pull out hierarchical data into a structure. Previous label is carried along to merge analyses of the same section.

regparser.notice.sxs.**find_page**(*xml*, *index_line*, *page_number*)
Find the FR page that includes the indexed line

regparser.notice.sxs.**find_section_by_section**(*xml_tree*)
Find the section-by-section analysis of this notice

regparser.notice.sxs.**is_backtrack**(*previous_label*, *next_label*)
If we've already processes a header with 22(c) in it, we can assume that any following headers with 1111.22 are *not* supposed to be an analysis of 1111.22

regparser.notice.sxs.**is_child_of**(*child_xml*, *header_xml*, *cfr_part*, *header_citations=None*)
Children are paragraphs, have lower 'source', the header has citations and the child does not, the citations for header and child are the same or the citation in a child is incorrect

regparser.notice.sxs.**parse_into_labels**(*txt*, *part*)
Find what part+section+(paragraph) (could be multiple) this text is related to.

regparser.notice.sxs.**remove_extract**(*xml_tree*)
Occasionally, the paragraphs/etc. useful to us are inside an EXTRACT tag. To normalize, move everything in an EXTRACT tag out

regparser.notice.sxs.**split_into_ttsr**(*sxs*, *cfr_part*)
Split the provided list of xml nodes into a node with a title, a sequence of text nodes, a sequence of nodes associated with the sub sections of this header, and the remaining xml nodes

regparser.notice.util module

regparser.notice.util.**body_to_string**(*xml_node*)
Create a string from the text of this node and its children (without the outer tag)

regparser.notice.util.**prepost_pend_spaces**(*el*)
FR's XML doesn't always add spaces around tags that clearly need them. Account for this by adding spaces around the el where needed.

regparser.notice.util.**spaces_then_remove**(*el*, *tag_str*)
FR's XML tends to not add spaces where needed, which leads to the removal of tags sometimes smashing together words.

regparser.notice.util.**swap_emphasis_tags**(*el*)
FR's XML uses a different set of tags than the standard we'd like (XHTML). Swap out at needed

regparser.notice.xml module

Module contents

regparser.tree package

Subpackages

regparser.tree.appendix package

Submodules

regparser.tree.appendix.carving module

regparser.tree.appendix.carving.**find_appendix_start**(*text*)
Find the start of the appendix (e.g. Appendix A)

regparser.tree.appendix.generic module

regparser.tree.appendix.generic.**find_next_segment**(*text*)
Find the start/end of the next segment. A segment for the generic appendix parser is something separated by a title-ish line (a short line with title-case words).

regparser.tree.appendix.generic.**is_title_case**(*line*)
Determine if a line is title-case (i.e. the first letter of every word is upper-case. More readable than the equivalent all([]) form.

regparser.tree.appendix.generic.**segments**(*text*)
Return a list of segment offsets. See find_next_segment()

regparser.tree.appendix.tree module

Module contents

regparser.tree.depth package

Submodules

regparser.tree.depth.derive module

class regparser.tree.depth.derive.**ParAssignment**(*typ*, *idx*, *depth*)
Bases: tuple

depth
Alias for field number 2

idx
Alias for field number 1

typ
Alias for field number 0

class regparser.tree.depth.derive.**Solution**(*assignment*, *weight=1.0*)
Bases: object

A collection of assignments + a weight for how likely this solution is (after applying heuristics)

copy_with_penalty(*penalty*)
Immutable copy while modifying weight

pretty_str()

regparser.tree.depth.derive.**debug_idx**(*marker_list*, *constraints=None*)
Binary search through the markers to find the point at which derive_depths no longer works

regparser.tree.depth.derive.**derive_depths**(*original_markers*, *additional_constraints=None*)
Use constraint programming to derive the paragraph depths associated with a list of paragraph markers. Additional constraints (e.g. expected marker types, etc.) can also be added. Such constraints are functions of two parameters, the constraint function (problem.addConstraint) and a list of all variables

regparser.tree.depth.heuristics module

Set of heuristics for trimming down the set of solutions. Each heuristic works by penalizing a solution; it's then up to the caller to grab the solution with the least penalties.

regparser.tree.depth.heuristics.**prefer_diff_types_diff_levels**(*solutions*, *weight=1.0*)
Dock solutions which have different markers appearing at the same level. This also occurs, but not often.

regparser.tree.depth.heuristics.**prefer_multiple_children**(*solutions*, *weight=1.0*)
Dock solutions which have a paragraph with exactly one child. While this is possible, it's unlikely.

regparser.tree.depth.heuristics.**prefer_no_markerless_sandwich**(*solutions*, *weight=1.0*)
Prefer solutions which don't use MARKERLESS to switch depth, like a MARKERLESS

a

regparser.tree.depth.heuristics.**prefer_shallow_depths**(*solutions*, *weight=0.1*)
Dock solutions which have a higher maximum depth

regparser.tree.depth.markers module

Namespace for collecting the various types of markers

regparser.tree.depth.markers.**deemphasize**(*marker*)
Though the knowledge of emphasis is helpful for determining depth, it is _unhelpful_ in other scenarios, where we only care about the plain text. This function removes <E> tags

regparser.tree.depth.markers.**emphasize**(*marker*)
The final depth levels for regulation text are emphasized, so we keep their <E> tags to distinguish them from previous levels. This function will wrap a marker in an <E> tag

regparser.tree.depth.optional_rules module

Depth derivation has a mechanism for _optional_ rules. This module contains a collection of such rules. All functions should accept two parameters; the latter is a list of all variables in the system; the former is a function which can be used to constrain the variables. This allows us to define rules over subsets of the variables rather than all of them, should that make our constraints more useful

regparser.tree.depth.optional_rules.**depth_type_inverses**(*constrain*, *all_variables*)
If paragraphs are at the same depth, they must share the same type. If paragraphs are the same type, they must share the same depth

regparser.tree.depth.optional_rules.**limit_paragraph_types**(**p_types*)
Constraint paragraphs to a limited set of paragraph types. This can reduce the search space if we know (for example) that the text comes from regulations and hence does not have capitalized roman numerals

regparser.tree.depth.optional_rules.**limit_sequence_gap**(*size=0*)
We've loosened the rules around sequences of paragraphs so that paragraphs can be skipped. This allows arbitrary tightening of that rule, effectively allowing gaps of a limited size

regparser.tree.depth.optional_rules.**star_new_level**(*constrain*, *all_variables*)
STARS should never have subparagraphs as it'd be impossible to determine where in the hierarchy these subparagraphs belong. @todo: This _probably_ should be a general rule, but there's a test that this breaks in the interpretations. Revisit with CFPB regs

regparser.tree.depth.optional_rules.**stars_occupy_space**(*constrain*, *all_variables*)
Star markers can't be ignored in sequence, so 1, *, 2 doesn't make sense for a single level, unless it's an inline star. In the inline case, we can think of it as 1, intro-text-to-1, 2

regparser.tree.depth.pair_rules module

Rules relating to two paragraph markers in sequence. The rules are "positive" in the sense that each allows for a particular scenario (rather than denying all other scenarios). They combine in the eponymous function, where, if any of the rules return True, we pass. Otherwise, we fail.

class regparser.tree.depth.pair_rules.**MarkerAssignment**
Bases: *regparser.tree.depth.pair_rules.MarkerAssignment*

is_inline_stars()
Inline stars (* * *) often behave quite differently from both STARS and other markers.

is_markerless()
We will often check whether an assignment is MARKERLESS. This function makes that clearer

is_stars()
We will often check whether an assignment is either STARS or inline stars (* * *). This function makes that clearer

regparser.tree.depth.pair_rules.**continuing_seq**(*prev*, *curr*)
E.g. "d, e" is good, but "e, d" is not. We also want to allow some paragraphs to be skipped, e.g. "d, g"

regparser.tree.depth.pair_rules.**decreasing_stars**(*prev*, *curr*)
Two stars in a row can exist if the second is shallower than the first

regparser.tree.depth.pair_rules.**decrement_depth**(*prev*, *curr*)
Decrementing depth is okay unless we're using inline stars

regparser.tree.depth.pair_rules.**marker_star_level**(*prev*, *curr*)
Allow a marker to be followed by stars if those stars are deeper. If not inline, also allow the stars to be at the same depth

regparser.tree.depth.pair_rules.**markerless_same_level**(*prev*, *curr*)
Markerless paragraphs can be followed by any type on the same level as long as that's beginning a new sequence

regparser.tree.depth.pair_rules.**new_sequence**(*prev*, *curr*)
Allow depth to be incremented if starting a new sequence

regparser.tree.depth.pair_rules.**pair_rules**(*prev_typ*, *prev_idx*, *prev_depth*, *typ*, *idx*, *depth*)
Combine all of the above rules

regparser.tree.depth.pair_rules.**paragraph_markerless**(*prev*, *curr*)
A non-markerless paragraph followed by a markerless paragraph can be one level deeper

regparser.tree.depth.pair_rules.**same_level_stars**(*prev*, *curr*)
Two stars in a row can exist on the same level if the previous is inline

regparser.tree.depth.pair_rules.**star_marker_level**(*prev*, *curr*)
Allow markers to be on the same level as a preceding star

regparser.tree.depth.rules module

Namespace for constraints on paragraph depth discovery.

For the purposes of this module a "symmetry" refers to two perfectly valid solutions to a problem whose differences are irrelevant. For example, if the distinctions between a vs. a STARS STARS may not matter if we're planning to ignore the final STARS anyway. To "break" this symmetry, we explicitly reject one solution; this reduces the number of permutations we care about dramatically.

regparser.tree.depth.rules.**ancestors**(*all_prev*)
Given an assignment of values, construct a list of the relevant parents, e.g. 1, i, a, ii, A gives us 1, ii, A

regparser.tree.depth.rules.**continue_previous_seq**(*typ*, *idx*, *depth*, **all_prev*)
Constrain the current marker based on all markers leading up to it

regparser.tree.depth.rules.**depth_type_order**(*order*)
Create a function which constrains paragraphs depths to a particular type sequence. For example, we know a priori what regtext and interpretation markers' order should be. Adding this constrain speeds up solution finding.

regparser.tree.depth.rules.**marker_stars_markerless_symmetry**(*pprev_typ*, *pprev_idx*, *pprev_depth*, *prev_typ*, *prev_idx*, *prev_depth*, *typ*, *idx*, *depth*)

When we have the following symmetry: a a a

STARS vs. STARS vs. STARS MARKERLESS MARKERLESS MARKERLESS

Prefer the middle

regparser.tree.depth.rules.**markerless_stars_symmetry**(*pprev_typ*, *pprev_idx*, *pprev_depth*, *prev_typ*, *prev_idx*, *prev_depth*, *typ*, *idx*, *depth*)

Given MARKERLESS, STARS, MARKERLESS want to break these symmetries:

MARKERLESS MARKERLESS STARS vs. STARS MARKERLESS MARKERLESS

Here, we don't really care about the distinction, so we'll opt for the former.

regparser.tree.depth.rules.**must_be**(*value*)

A constraint that the given variable must matches the value.

regparser.tree.depth.rules.**same_parent_same_type**(**all_vars*)

All markers in the same parent should have the same marker type. Exceptions for:

STARS, which can appear at any level Sequences which _begin_ with markerless paragraphs

regparser.tree.depth.rules.**star_sandwich_symmetry**(*pprev_typ*, *pprev_idx*, *pprev_depth*, *prev_typ*, *prev_idx*, *prev_depth*, *typ*, *idx*, *depth*)

Symmetry breaking constraint that places STARS tag at specific depth so that the resolution of

c

? ? ? ? ? ? <- Potential STARS depths 5

can only be one of

OR

c c STARS STARS

5 5 Stars also cannot be used to skip a level (similar to markerless sandwich, above)

regparser.tree.depth.rules.**triplet_tests**(**triplet_seq*)

Run propositions around a sequence of three markers. We combine them here so that they act as a single constraint

regparser.tree.depth.rules.**type_match**(*marker*)

The type of the associated variable must match its marker. Lambda explanation as in the above rule.

Module contents

regparser.tree.xml_parser package

Submodules

regparser.tree.xml_parser.appendices module

regparser.tree.xml_parser.extended_preprocessors module

regparser.tree.xml_parser.flatsubtree_processor module

class regparser.tree.xml_parser.flatsubtree_processor.**FlatParagraphProcessor**

Bases: *regparser.tree.xml_parser.paragraph_processor.ParagraphProcessor*

Paragraph Processor which does not try to derive paragraph markers

MATCHERS = [<regparser.tree.xml_parser.paragraph_processor.StarsMatcher object>, <regparser.tree.xml_parser.paragr

class regparser.tree.xml_parser.flatsubtree_processor.**FlatsubtreeMatcher**(*tags*, *node_type=u'regtext'*)

Bases: *regparser.tree.xml_parser.paragraph_processor.BaseMatcher*

Detects tags passed to it on init and processes them with the FlatParagraphProcessor. Also optionally sets node_type.

derive_nodes(*xml*, *processor=None*)

matches(*xml*)

regparser.tree.xml_parser.import_category module

class regparser.tree.xml_parser.import_category.**ImportCategoryMatcher**

Bases: *regparser.tree.xml_parser.paragraph_processor.BaseMatcher*

The IMPORTCATEGORY gets converted into a subtree with an appropriate title and unique paragraph marker

CATEGORY_RE = <_sre.SRE_Pattern object>

derive_nodes(*xml*, *processor=None*)

Finds and deletes the category header before recursing. Adds this header as a title.

classmethod marker(*header_text*)

Derive a unique, repeatable identifier for this subtree. This allows the same category to be reordered (e.g. if a note has been added), or a header with multiple reserved categories to be split (which would also re-order the categories that followed)

matches(*xml*)

regparser.tree.xml_parser.interpretations module

regparser.tree.xml_parser.paragraph_processor module

class regparser.tree.xml_parser.paragraph_processor.**BaseMatcher**

Bases: `object`

Base class defining the interface of various XML node matchers

derive_nodes(*xml*, *processor=None*)

Given an xml node which this matcher applies against, convert it into a list of Node structures. *processor* is the paragraph processor which we are being executed in. May be useful when determining how to create the Nodes

matches(*xml*)

Test the xml element – does this matcher apply?

class regparser.tree.xml_parser.paragraph_processor.**FencedMatcher**

Bases: *regparser.tree.xml_parser.paragraph_processor.BaseMatcher*

Use github-like fencing to indicate this is code

derive_nodes(*xml*, *processor=None*)

matches(*xml*)

class regparser.tree.xml_parser.paragraph_processor.**GraphicsMatcher**

Bases: *regparser.tree.xml_parser.paragraph_processor.BaseMatcher*

Convert Graphics tags into a markdown-esque format

derive_nodes(*xml*, *processor=None*)

matches(*xml*)

class regparser.tree.xml_parser.paragraph_processor.**HeaderMatcher**
Bases: *regparser.tree.xml_parser.paragraph_processor.BaseMatcher*

derive_nodes(*xml*, *processor=None*)

matches(*xml*)

class regparser.tree.xml_parser.paragraph_processor.**IgnoreTagMatcher**(**tags*)
Bases: *regparser.tree.xml_parser.paragraph_processor.SimpleTagMatcher*

As we log warnings when we don't know how to process a tag, this matcher allows us to positively acknowledge that we're ignoring some matches

derive_nodes(*xml*, *processor=None*)

class regparser.tree.xml_parser.paragraph_processor.**ParagraphProcessor**
Bases: object

Processing paragraphs in a generic manner requires a lot of state to be carried in between xml nodes. Use a class to wrap that state so we can compartmentalize processing with various tags. This is an abstract class; regtext, interpretations, appendices, etc. should inherit and override where needed

DEPTH_HEURISTICS = OrderedDict([(<function prefer_diff_types_diff_levels>, 0.8), (<function prefer_multiple_childre

MATCHERS = []

static additional_constraints()
Hook for subtypes to add additional constraints

build_hierarchy(*root*, *nodes*, *depths*)
Given a root node, a flat list of child nodes, and a list of depths, build a node hierarchy around the root

carry_label_to_children(*node*)
Takes a node and recursively processes its children to add the appropriate label prefix to them.

parse_nodes(*xml*)
Derive a flat list of nodes from this xml chunk. This does nothing to determine node depth

process(*xml*, *root*)

static relaxed_constraints()
Hook for subtypes to add relaxed constraints for retry logic

static replace_markerless(*stack*, *node*, *depth*)
Assign a unique index to all of the MARKERLESS paragraphs

select_depth(*depths*)
There might be multiple solutions to our depth processing problem. Use heuristics to select one.

static separate_intro(*nodes*)
In many situations the first unlabeled paragraph is the "intro" text for a section. We separate that out here

class regparser.tree.xml_parser.paragraph_processor.**SimpleTagMatcher**(**tags*)
Bases: *regparser.tree.xml_parser.paragraph_processor.BaseMatcher*

Simple example tag matcher – it listens for specific tags and derives a single node with the associated body

derive_nodes(*xml*, *processor=None*)

matches(*xml*)

class regparser.tree.xml_parser.paragraph_processor.**StarsMatcher**
Bases: *regparser.tree.xml_parser.paragraph_processor.BaseMatcher*

<STARS> indicates a chunk of text which is being skipped over

derive_nodes(*xml*, *processor=None*)

matches(*xml*)

class regparser.tree.xml_parser.paragraph_processor.**TableMatcher**
Bases: *regparser.tree.xml_parser.paragraph_processor.BaseMatcher*

Matches the GPOTABLE tag

derive_nodes(*xml*, *processor=None*)

matches(*xml*)

regparser.tree.xml_parser.preprocessors module

Set of transforms we run on notice XML to account for common inaccuracies in the XML

class regparser.tree.xml_parser.preprocessors.**ApprovalsFP**
Bases: *regparser.tree.xml_parser.preprocessors.PreProcessorBase*

We expect certain text to an APPRO tag, but it is often mistakenly found inside FP tags. We use REGEX to determine which nodes need to be fixed.

REGEX = <_sre.SRE_Pattern object at 0x325b600>

static strip_extracts(*xml*)
APPROs should not be alone in an EXTRACT

transform(*xml*)

class regparser.tree.xml_parser.preprocessors.**ExtractTags**
Bases: *regparser.tree.xml_parser.preprocessors.PreProcessorBase*

Often, what should be a single EXTRACT tag is broken up by incorrectly positioned subtags. Try to find any such EXTRACT sandwiches and merge.

FILLING = (u'FTNT', u'GPOTABLE')

combine_with_following(*extract*, *include_tag*)
We need to merge an extract with the following tag. Rather than iterating over the node, text, tail text, etc. we're taking a more naive solution: convert to a string, reparse

extract_pair(*extract*)
Checks for and merges two EXTRACT tags in sequence

sandwich(*extract*)
Checks for this pattern: EXTRACT FILLING EXTRACT, and, if present, combines the first two tags. The two EXTRACTs would get merged in a later pass

static strip_root_tag(*string*)

transform(*xml*)

class regparser.tree.xml_parser.preprocessors.**Footnotes**
Bases: *regparser.tree.xml_parser.preprocessors.PreProcessorBase*

The XML separates the content of footnotes and where they are referenced. To make it more semantic (and easier to process), we find the relevant footnote and attach its text to the references. We also need to split references apart if multiple footnotes apply to the same <SU>

IS_REF_PREDICATE = u'not(ancestor::TNOTE) and not(ancestor::FTNT)'

XPATH_FIND_NOTE_TPL = u"./following::SU[(ancestor::TNOTE or ancestor::FTNT) and text()='{0}']"

XPATH_IS_REF = u'.//SU[not(ancestor::TNOTE) and not(ancestor::FTNT)]'

add_ref_attributes(*xml*)
Modify each footnote reference so that it has an attribute containing its footnote content

static **is_reasonably_close**(*referencing*, *referenced*)
We want to make sure that _potential_ footnotes are truly related, as SU might also indicate generic superscript. To match a footnote with its content, we'll try to find a common SECTION ancestor. We'll also consider the two SUs related if neither has a SECTION ancestor, though we might want to restrict this further in the future.

split_comma_footnotes(*xml*)
Convert XML such as <SU>1, 2, 3</SU> into distinct SU elements: <SU>1</SU> <SU>2</SU> <SU>3</SU> for easier reference

transform(*xml*)

class regparser.tree.xml_parser.preprocessors.**ImportCategories**
Bases: *regparser.tree.xml_parser.preprocessors.PreProcessorBase*

447.21 contains an import list, but the XML doesn't delineate the various categories well. We've created *IMPORTCATEGORY* tags to handle the hierarchy correctly, but we need to modify the XML to insert them in appropriate locations

CATEGORY_HD = u".//HD[contains(., 'categor')]"

SECTION_HD = u"//SECTNO[contains(., '447.21')]"

static **remove_extract**(*section*)
The XML currently (though this may change) contains a semantically meaningless EXTRACT. Remove it

static **split_categories**(*category_headers*)
We now have a big chunk of flat XML with headers and paragraphs. We'll make it semantic by converting these into bundles and wrapping them in IMPORTCATEGORY tags

transform(*xml*)

class regparser.tree.xml_parser.preprocessors.**PreProcessorBase**
Bases: object

Base class for all the preprocessors. Defines the interface they must implement

transform(*xml*)
Transform the input xml. Mutates that xml, so be sure to make a copy if needed

regparser.tree.xml_parser.preprocessors.**atf_i50031**(*xml*)
478.103 also contains a shorter form, which appears in a smaller poster. Unfortunately, the XML didn't include the appropriate NOTE inside the corresponding EXTRACT

regparser.tree.xml_parser.preprocessors.**atf_i50032**(*xml*)
478.103 contains a chunk of text which is meant to appear in a poster and be easily copy-paste-able. Unfortunately, the XML post 2003 isn't structured to contain all of the appropriate elements within the EXTRACT associated with the poster. This PreProcessor moves these additional elements back into the appropriate EXTRACT.

regparser.tree.xml_parser.preprocessors.**move_adjoining_chars**(*xml*)
If an e tag has an emdash or period after it, put the char inside the e tag

regparser.tree.xml_parser.preprocessors.**move_last_amdpar**(*xml*)
If the last element in a section is an AMDPAR, odds are the authors intended it to be associated with the following section

regparser.tree.xml_parser.preprocessors.**move_subpart_into_contents**(*xml*)
Account for SUBPART tags being outside their intended CONTENTS

regparser.tree.xml_parser.preprocessors.**parentheses_cleanup**(*xml*)
Clean up where parentheses exist between paragraph an emphasis tags

regparser.tree.xml_parser.preprocessors.**preprocess_amdpars**(*xml*)
Modify the AMDPAR tag to contain an <EREGS_INSTRUCTIONS> element. This element contains an interpretation of the AMDPAR, as viewed as a sequence of actions for how to modify the CFR. Do _not_ modify any existing EREGS_INSTRUCTIONS (they've been manually created)

regparser.tree.xml_parser.preprocessors.**promote_nested_tags**(*tag*, *xml*)
We don't currently support certain tags nested inside subparts, so promote each up one level

regparser.tree.xml_parser.preprocessors.**replace_html_entities**(*xml_bin_str*)
XML does not contain entity references for many HTML entities, yet the Federal Register XML sometimes contains the HTML entities. Replace them here, lest we throw off XML parsing

regparser.tree.xml_parser.reg_text module

regparser.tree.xml_parser.simple_hierarchy_processor module

class regparser.tree.xml_parser.simple_hierarchy_processor.**DepthParagraphMatcher**
Bases: *regparser.tree.xml_parser.paragraph_processor.BaseMatcher*

Convert a paragraph with an optional prefixing paragraph marker into an appropriate node. Does not know about collapsed markers nor most types of nodes.

derive_nodes(*xml*, *processor=None*)

matches(*xml*)

class regparser.tree.xml_parser.simple_hierarchy_processor.**SimpleHierarchyMatcher**(*tags*, *node_type*)
Bases: *regparser.tree.xml_parser.paragraph_processor.BaseMatcher*

Detects tags passed to it on init and converts the contents of any matches into a hierarchy based on the SimpleHierarchyProcessor. Sets the node_type of the subtree's root

derive_nodes(*xml*, *processor=None*)

matches(*xml*)

class regparser.tree.xml_parser.simple_hierarchy_processor.**SimpleHierarchyProcessor**
Bases: *regparser.tree.xml_parser.paragraph_processor.ParagraphProcessor*

ParagraphProcessor which attempts to pull out whatever paragraph marker is available and derive a hierarchy from that.

MATCHERS = [<regparser.tree.xml_parser.simple_hierarchy_processor.DepthParagraphMatcher object>]

additional_constraints()

regparser.tree.xml_parser.tree_utils module

class regparser.tree.xml_parser.tree_utils.**NodeStack**
Bases: *regparser.tree.priority_stack.PriorityStack*

The NodeStack aids our construction of a struct.Node tree. We process xml one paragraph at a time; using a priority stack allows us to insert items at their proper depth and unwind the stack (collecting children) as necessary

collapse()
After all of the nodes have been inserted at their proper levels, collapse them into a single root node

unwind()
Unwind the stack, collapsing sub-paragraphs that are on the stack into the children of the previous level.

regparser.tree.xml_parser.tree_utils.**footnotes_to_plaintext**(*node*, *add_spaces*)

regparser.tree.xml_parser.tree_utils.**get_node_text**(*node*, *add_spaces=False*)
Extract all the text from an XML node (including the text of it's children).

regparser.tree.xml_parser.tree_utils.**get_node_text_tags_preserved**(*xml_node*)
Get the body of an XML node as a string, avoiding a specific blacklist of bad tags.

regparser.tree.xml_parser.tree_utils.**prepend_parts**(*parts_prefix*, *n*)
Recursively preprend parts_prefix to the parts of the node n. Parts is a list of markers that indicates where you are in the regulation text.

regparser.tree.xml_parser.tree_utils.**replace_xml_node_with_text**(*node*, *text*)
There are some complications w/ lxml when determining where to add the replacement text. Account for all of that here.

regparser.tree.xml_parser.tree_utils.**replace_xpath**(*xpath*)
Decorator to convert all elements matching the provided xpath in to plain text. This'll convert the wrapped function into a new function which will search for the provided xpath and replace all matches

regparser.tree.xml_parser.tree_utils.**split_text**(*text*, *tokens*)
Given a body of text that contains tokens, splice the text along those tokens.

regparser.tree.xml_parser.tree_utils.**subscript_to_plaintext**(*node*, *add_spaces*)

regparser.tree.xml_parser.tree_utils.**superscript_to_plaintext**(*node*, *add_spaces*)

regparser.tree.xml_parser.us_code module

class regparser.tree.xml_parser.us_code.**USCodeMatcher**
Bases: *regparser.tree.xml_parser.paragraph_processor.BaseMatcher*

Matches a custom *USCODE* tag and parses it's contents with the USCodeProcessor. Does not use a custom node type at the moment

derive_nodes(*xml*, *processor=None*)

matches(*xml*)

class regparser.tree.xml_parser.us_code.**USCodeParagraphMatcher**
Bases: *regparser.tree.xml_parser.paragraph_processor.BaseMatcher*

Convert a paragraph found in the US Code into appropriate Nodes

derive_nodes(*xml*, *processor=None*)

matches(*xml*)

paragraph_markers(*text*)
We can't use tree_utils.get_paragraph_markers as that makes assumptions about the order of paragraph markers (specifically that the markers will match the order found in regulations). This is simpler, looking only at multiple markers at the beginning of the paragraph

class regparser.tree.xml_parser.us_code.**USCodeProcessor**
Bases: *regparser.tree.xml_parser.paragraph_processor.ParagraphProcessor*

ParagraphProcessor which converts a chunk of XML into Nodes. Only processes P nodes and limits the type of paragraph markers to those found in US Code

MATCHERS = [<regparser.tree.xml_parser.us_code.USCodeParagraphMatcher object>]

additional_constraints()

regparser.tree.xml_parser.xml_wrapper module

class regparser.tree.xml_parser.xml_wrapper.**XMLWrapper**(*xml*, *source=None*)
Bases: object

Wrapper around XML which provides a consistent interface shared by both Notices and Annual editions of XML

preprocess()
Unfortunately, the notice xml is often inaccurate. This function attempts to fix some of those (general) flaws. For specific issues, we tend to instead use the files in settings.LOCAL_XML_PATHS

xml_str()

xpath(**args*, ***kwargs*)

Module contents

Submodules

regparser.tree.build module

regparser.tree.interpretation module

regparser.tree.paragraph module

class regparser.tree.paragraph.**ParagraphParser**(*p_regex*, *node_type*)

best_start(*text*, *p_level*, *paragraph*, *starts*, *exclude=None*)
Given a list of potential paragraph starts, pick the best based on knowledge of subparagraph structure. Do this by checking if the id following the subparagraph (e.g. ii) is between the first match and the second. If so, skip it, as that implies the first match was a subparagraph.

build_tree(*text*, *p_level=0*, *exclude=None*, *label=None*, *title=''*)
Build a dict to represent the text hierarchy.

find_paragraph_start_match(*text*, *p_level*, *paragraph*, *exclude=None*)
Find the positions for the start and end of the requested label. p_Level is one of 0,1,2,3; paragraph is the index within that label. Return None if not present. Does not return results in the exclude list (a list of start/stop indices).

static **matching_subparagraph_ids**(*p_level*, *paragraph*)
Return a list of matches if this paragraph id matches one of the subparagraph ids (e.g. letter (i) and roman numeral (i).

paragraph_offsets(*text*, *p_level*, *paragraph*, *exclude=None*)
Find the start/end of the requested paragraph. Assumes the text does not just up a p_level – see build_paragraph_tree below.

paragraphs(*text*, *p_level*, *exclude=None*)
Return a list of paragraph offsets defined by the level param.

regparser.tree.paragraph.**hash_for_paragraph**(*text*)
Hash a chunk of text and convert it into an integer for use with a MARKERLESS paragraph identifier. We'll trim to just 8 hex characters for legibility. We don't need to fear hash collisions as we'll have 16**8 ~ 4 billion possibilities. The birthday paradox tells us we'd only expect collisions after ~ 60 thousand entries. We're expecting at most a few hundred

regparser.tree.paragraph.**p_level_of**(*marker*)
Given a marker(string), determine the possible paragraph levels it could fall into. This is useful for determining the order of paragraphs

regparser.tree.priority_stack module

class regparser.tree.priority_stack.**PriorityStack**
Bases: object

add(*node_level*, *node*)
Add a new node with level node_level to the stack. Unwind the stack when necessary. Returns *self* for chaining

lineage()
Fetch the last element of each level of priorities. When the stack is used to keep track of a tree, this list includes a list of 'parents', as the last element of each level is the parent being processed.

lineage_with_level()

peek()

peek_last()

peek_level(*level*)
Find a whole level of nodes in the stack

pop()

push(*m*)

push_last(*m*)

size()

unwind()
Combine nodes as needed while walking back up the stack. Intended to be overridden, as how to combine elements depends on the element type.

regparser.tree.reg_text module

regparser.tree.reg_text.**build_empty_part**(*part*)
When a regulation doesn't have a subpart, we give it an emptypart (a dummy subpart) so that the regulation tree is consistent.

regparser.tree.reg_text.**build_subjgrp**(*title*, *part*, *letter_list*)
We're constructing a fake "letter" here by taking the first letter of each word in the subjgrp's title, or using the first two letters of the first word if there's just one—we're avoiding single letters to make sure we don't duplicate an existing subpart, and we're hoping that the initialisms created by this method are unique for this regulation. We can make this more robust by accepting a list of existing initialisms and returning both that list and the Node, and checking against the list as we construct them.

regparser.tree.reg_text.**build_subpart**(*text*, *part*)

regparser.tree.reg_text.**find_next_section_start**(*text*, *part*)
Find the start of the next section (e.g. 205.14)

regparser.tree.reg_text.**find_next_subpart_start**(*text*)
Find the start of the next Subpart (e.g. Subpart B)

regparser.tree.reg_text.**next_section_offsets**(*text*, *part*)
Find the start/end of the next section

regparser.tree.reg_text.**next_subpart_offsets**(*text*)
Find the start,end of the next subpart

regparser.tree.reg_text.**sections**(*text*, *part*)
Return a list of section offsets. Does not include appendices.

regparser.tree.reg_text.**subjgrp_label**(*starting_title*, *letter_list*)

regparser.tree.reg_text.**subparts**(*text*)
Return a list of subpart offset. Does not include appendices, supplements.

regparser.tree.struct module

class regparser.tree.struct.**FrozenNode**(*text=''*, *children=()*, *label=()*, *title=''*, *node_type=u'regtext'*, *tagged_text=''*)
Bases: object

Immutable interface for nodes. No guarantees about internal state.

child_labels

children

clone(***kwargs*)
Implement a namedtuple *_replace* style functionality, copying all fields that aren't explicitly replaced.

static from_node(*node*)
Convert a struct.Node (or similar) into a struct.FrozenNode. This also checks if this node has already been instantiated. If so, it returns the instantiated version (i.e. only one of each identical node exists in memory)

hash

label

label_id

node_type

prototype()
When we instantiate a FrozenNode, we add it to _pool if we've not seen an identical FrozenNode before. If we have, we want to work with that previously seen version instead. This method returns the _first_ FrozenNode with identical fields

tagged_text

text

title

class regparser.tree.struct.**FullNodeEncoder**(*skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *encoding='utf-8'*, *default=None*)

Bases: json.encoder.JSONEncoder

Encodes Nodes into JSON, not losing any of the fields

FIELDS = set(['tagged_text', 'title', 'text', 'source_xml', 'label', 'node_type', 'children'])

default(*obj*)

class regparser.tree.struct.**Node**(*text=''*, *children=None*, *label=None*, *title=None*, *node_type=u'regtext'*, *source_xml=None*, *tagged_text=''*)

Bases: object

APPENDIX = u'appendix'

EMPTYPART = u'emptypart'

EXTRACT = u'extract'

INTERP = u'interp'

INTERP_MARK = 'Interp'

MARKERLESS_REGEX = <_sre.SRE_Pattern object>

NOTE = u'note'

REGTEXT = u'regtext'

SUBPART = u'subpart'

cfr_part

depth()

Inspect the label and type to determine the node's depth

is_markerless()

classmethod is_markerless_label(*label*)

is_section()

Sections are contained within subparts/subject groups. They are not part of the appendix

label_id()

walk(*fn*)

See walk(node, fn)

class regparser.tree.struct.**NodeEncoder**(*skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *encoding='utf-8'*, *default=None*)

Bases: json.encoder.JSONEncoder

Custom JSON encoder to handle Node objects

default(*obj*)

regparser.tree.struct.**filter_walk**(*node*, *fn*)

Perform fn on the label for every node in the tree and return a list of nodes on which the function returns truthy.

regparser.tree.struct.**find**(*root*, *label*)
Search through the tree to find the node with this label.

regparser.tree.struct.**find_first**(*root*, *predicate*)
Walk the tree and find the first node which matches the predicate

regparser.tree.struct.**find_parent**(*root*, *label*)
Search through the tree to find the _parent_ or a node with this label.

regparser.tree.struct.**frozen_node_decode_hook**(*d*)
Convert a JSON object into a FrozenNode

regparser.tree.struct.**full_node_decode_hook**(*d*)
Convert a JSON object into a full Node

regparser.tree.struct.**merge_duplicates**(*nodes*)
Given a list of nodes with the same-length label, merge any duplicates (by combining their children)

regparser.tree.struct.**treeify**(*nodes*)
Given a list of nodes, convert those nodes into the appropriate tree structure based on their labels. This assumes that all nodes will fall under a set of 'root' nodes, which have the min-length label.

regparser.tree.struct.**walk**(*node*, *fn*)
Perform fn for every node in the tree. Pre-order traversal. fn must be a function that accepts a root node.

regparser.tree.supplement module

regparser.tree.supplement.**find_supplement_start**(*text*, *supplement='I'*)
Find the start of the supplement (e.g. Supplement I)

Module contents

Submodules

regparser.api_stub module

regparser.api_writer module

class regparser.api_writer.**APIWriteContent**(**path_parts*)
This writer writes the contents to the specified API

write(*python_obj*)
Write the object (as json) to the API

class regparser.api_writer.**AmendmentNodeEncoder**(*skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *encoding='utf-8'*, *default=None*)
Bases: *regparser.notice.encoder.AmendmentEncoder*, *regparser.tree.struct.NodeEncoder*

class regparser.api_writer.**Client**(*base*)
A Client for writing regulation(s) and meta data.

diff(*label*, *old_version*, *new_version*)

layer(*layer_name*, *doc_type*, *doc_id*)

notice(*doc_number*)

preamble(*doc_number*)

regulation(*label*, *doc_number*)

class regparser.api_writer.**FSWriteContent**(**path_parts*)
This writer places the contents in the file system

write(*python_obj*)
Write the object as json to disk

class regparser.api_writer.**GitWriteContent**(**path_parts*)
This writer places the content in a git repo on the file system

static folder_name(*node*)
Directories are generally just the last element a node's label, but subparts and interpretations are a little special.

write(*python_object*)

write_tree(*root_path*, *node*)
Given a file system path and a node, write the node's contents and recursively write its children to the provided location.

regparser.builder module

regparser.citations module

class regparser.citations.**Label**(*schema=None*, ***kwargs*)
Bases: object

SCHEMA_FIELDS = set(['p2', 'p3', 'p1', 'p6', 'p7', 'p4', 'p5', 'cfr_title', 'p8', 'p9', 'comment', 'appendix', 'appendix_sect

app_schema = ('part', 'appendix', 'p1', 'p2', 'p3', 'p4', 'p5', 'p6', 'p7', 'p8', 'p9')

app_sect_schema = ('part', 'appendix', 'appendix_section', 'p1', 'p2', 'p3', 'p4', 'p5', 'p6', 'p7', 'p8', 'p9')

comment_schema = ('comment', 'c1', 'c2', 'c3', 'c4')

copy(*schema=None*, ***kwargs*)
Keep any relevant prefix when copying

default_schema = ('cfr_title', 'part', 'section', 'p1', 'p2', 'p3', 'p4', 'p5', 'p6', 'p7', 'p8', 'p9')

static determine_schema(*settings*)

classmethod from_node(*node*)
Convert between a struct.Node and a Label; use heuristics to determine which schema to follow. Node labels aren't as expressive as Label objects

labels_until(*other*)
Given *self* as a starting point and *other* as an end point, yield a *Label* for paragraphs in between. For example, if *self* is something like 123.45(a)(2) and end is 123.45(a)(6), this should emit 123.45(a)(3), (4), and (5)

regtext_schema = ('cfr_title', 'part', 'section', 'p1', 'p2', 'p3', 'p4', 'p5', 'p6', 'p7', 'p8', 'p9')

to_list(*for_node=True*)
Convert a Label into a struct.Node style label list. Node labels don't contain CFR titles

class regparser.citations.**ParagraphCitation**(*start*, *end*, *label*, *full_start=None*, *full_end=None*, *in_clause=False*)
Bases: object

regparser.citations.**cfr_citations**(*text*, *include_fill=False*)
Find all citations which include CFR title and part

regparser.citations.**internal_citations**(*text*, *initial_label=None*, *require_marker=False*, *title=None*)
List of all internal citations in the text. require_marker helps by requiring text be prepended by 'comment'/'paragraphs'/etc. title represents the CFR title (e.g. 11 for FEC, 12 for CFPB regs) and is used to correctly parse citations of the the form 11 CFR 110.1 when 11 CFR 110 is the regulation being parsed.

regparser.citations.**match_to_label**(*match*, *initial_label*, *comment=False*)
Return the citation and offsets for this match

regparser.citations.**multiple_citations**(*matches*, *initial_label*, *comment=False*, *include_fill=False*)
Similar to single_citations save that we have a compound citation, such as "paragraphs (b), (d), and (f). Yield a ParagraphCitation for each sub-citation. We refer to the first match as "head" and all following as "tail"

regparser.citations.**remove_citation_overlaps**(*text*, *possible_markers*)
Given a list of markers, remove any that overlap with citations

regparser.citations.**select_encompassing_citations**(*citations*)
The same citation might be found by multiple grammars; we take the most-encompassing of any overlaps

regparser.citations.**single_citations**(*matches*, *initial_label*, *comment=False*)
For each pyparsing match, yield the corresponding ParagraphCitation

regparser.content module

We need to modify content from time to time, e.g. image overrides and xml macros. To provide flexibility in future expansion, we provide a layer of indirection here.

TODO: Delete and replace with plugins.

class regparser.content.**ImageOverrides**
Bases: object

static get(*key*, *default=None*)

class regparser.content.**Macros**
Bases: object

regparser.federalregister module

regparser.search module

regparser.search.**find_offsets**(*text*, *search_fn*)
Find the start and end of an appendix, supplement, etc.

regparser.search.**find_start**(*text*, *heading*, *index*)
Find the start of an appendix, supplement, etc.

regparser.search.**segments**(*text*, *offsets_fn*, *exclude=None*)

Split a block of text into a list of its sub parts. Often this means calling the offsets function repeatedly until there is no more text to process.

regparser.utils module

Module contents

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

Python Module Index

r

Index

D

E

F

G

M

N

O

P

Q

R

S