educe Documentation

Release 0.1

Eric Kow

Nov 27, 2017

Contents

Contents:

CHAPTER 1

User manual

Educe is mainly a library but it comes with a small number of command line tools that can be useful for poking and prodding at the corpora that it supports

1.1 STAC tools

Educe comes with a number of command line utilities for querying, checking, and modifying the STAC corpus:

- stac-util: queries
- stac-check: sanity checks (development)
- stac-edit: modifications to (development)
- stac-oneoff: rare modifications (development)

The first tool (stac-util) may be useful to all users of the STAC corpus, whereas the last three (stac-check, stac-edit, and stac-oneoff) may be more of interest for corpus development work.

1.1.1 stac-util

The stac-util toolkit provides some potentially useful queries on the corpus.

stac-util text

Dump the text in documents along with segment annotations

```
stac-util text --doc s2-leagueM-game2\
    --subdoc 02 --anno 'BRONZE|SILVER|GOLD' --stage discourse
```

This utility can be useful for getting a sense for what a particular document contains, without having to fire up the Glozz platform

```
========== s2-leagueM-game2 [02] discourse SILVER ============

72 : gotwood4sheep : [anyone got wood?]
73 : gotwood4sheep : [i can offer sheep]
74 : gotwood4sheep : [phrased in such a way i don't riff on my un]
75 : inca : [i'm up for that]
76 : CheshireCatGrin : [I have no wood]
77 : gotwood4sheep : [1:1?]
78 : inca : [yep,] [only got one]
81 : gotwood4sheep : [matt, do you got clay?]  [I can offer many things]
82 : CheshireCatGrin : [No clay either]
83 : gotwood4sheep : [anyone else?]
84 : dmm : [i think clay is in short supply]
85 : inca : [sorry,] [none here either]
86 : gotwood4sheep : [indeed, something to do with a robber on the 5]
87 : gotwood4sheep : [alas]
```

stac-util count

Display some basic counts on the corpus or a given subset thereof

```
stac-util count --doc s1-league3-game4
```

The output includes the number of instances of EDUs, turns, etc

```
Document structure
============================================================

per doc      total    min    max    mean    median
---------  -------  -----  -----  ------  --------
doc              1
subdoc           3      3      3       3         3
dialogue         7      7      7       7         7
turn star       25     25     25      25        25
turn            28     28     28      28        28
edu             58     58     58      58        58
...
```

along with dialogue-acts and relation instances...

```
Relation instances
============================================================

BRONZE                  total
--------------------  -------
Comment                     3
Elaboration                 1
Acknowledgement             4
Continuation                4
Explanation                 1
Q-Elab                      3
Result                      3
Background                  1
Parallel                    2
Question-answer_pair        8
TOTAL                      30
...
```

stac-util count-rfc

Count right frontier violations given all the RFC algorithms we have implemented:

```
stac-util count-rfc --doc pilot21
```

Output for the above includes both a total count and a pers label count

```
Both                    total    basic    mlast
--------------------  -------  -------  -------
TOTAL                     290       33       11
Question-answer_pair       91        4        0
Comment                    32        7        5
Continuation               23        3        1
Elaboration                22        4        0
Q-Elab                     22        3        1
Acknowledgement            20        2        0
...
```

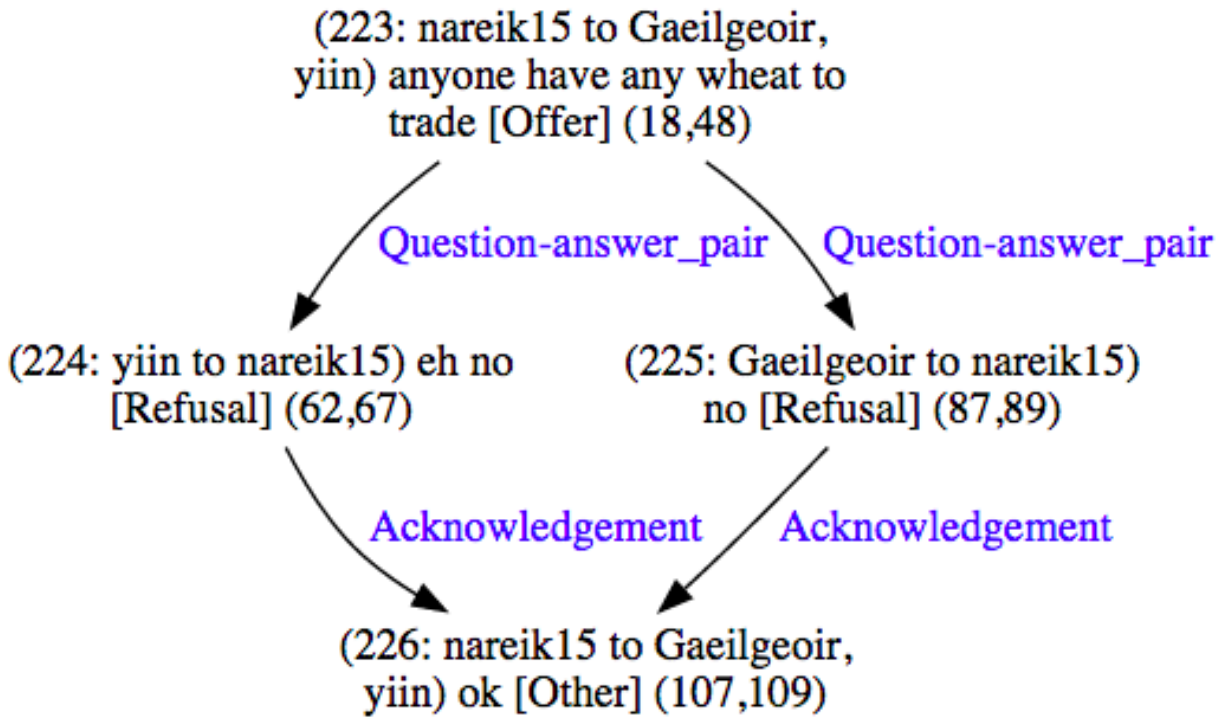
stac-util count-shapes

Count and draw the number of instances of shapes that we deem to be interesting (for now, this only means "lozenges", but we may come up with other shapes in the future, for example, instances of nodes with in-degree > 1)

```
stac-util count-shapes --anno 'GOLD|SILVER|BRONZE'\
    --output /tmp/graphs\
    data/socl-season1
```

Aside from the graph below, this displays a per-document count along with the total

```
s1-league2-game1 [14] discourse SILVER 1 (4)
s1-league2-game2 [01] discourse GOLD 3 (23)
s1-league2-game2 [02] discourse GOLD 1 (5)
s1-league2-game2 [03] discourse GOLD 1 (6)
s1-league2-game3 [03] discourse BRONZE 2 (10)
s1-league2-game4 [01] discourse BRONZE 1 (4)
s1-league2-game4 [03] discourse BRONZE 1 (6)
...
TOTAL lozenges: 46
TOTAL edges in lozenges: 234
```



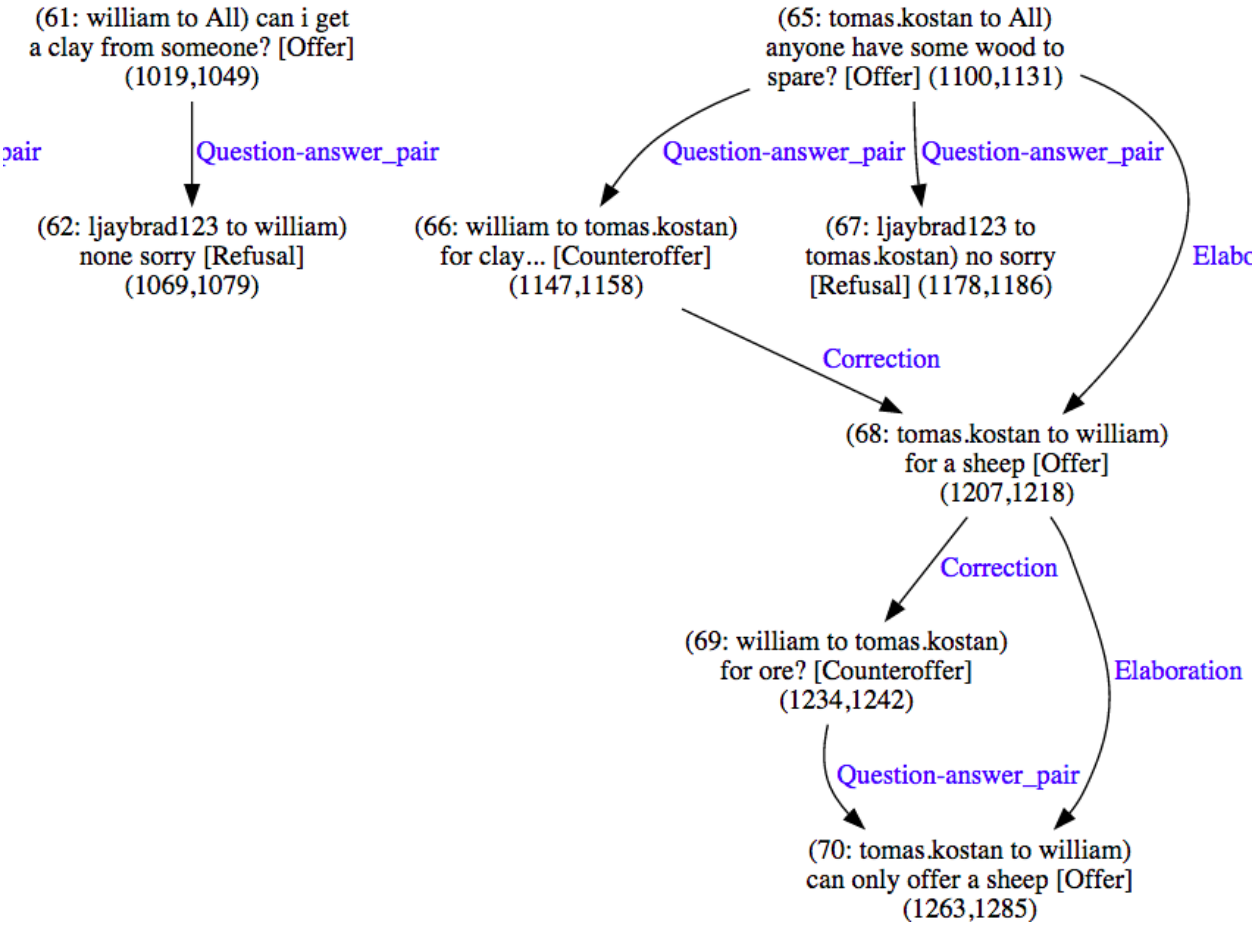
stac-util graph

Draw the discourse graph for a corpus

```
stac-util graph --doc s1-league1-game2 --anno SILVER\
    --output /tmp/graphs\
    data/socl-season1
```

Tips:

- *–strip-cdus* shows what the graph would look like with an automated CDU-removing algorithm applied to it
- *–rfc <algo>* will highlight the right frontier and violations given an RFC algorithm (eg *–rfc basic*)

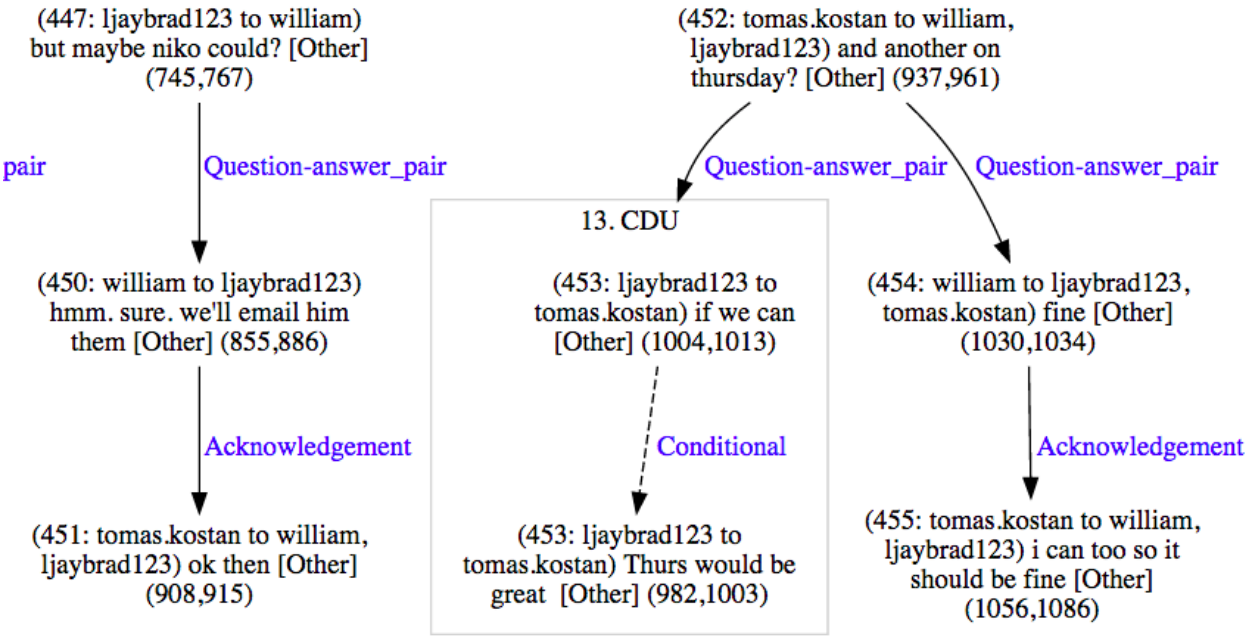


stac-util filter-graph

View all instances of a relation (or set of relations)

```
stac-util filter-graph --doc s1-league1-game2\
    --output /tmp/graphs\
    data/socl-season1\
    Question-answer_pair Acknowledgement
```

(Sorry, easy mode not available)



1.1.2 stac-check

The STAC corpus (at the time of this writing 2015-06-12) is a work in progress, and so some of our utilities are geared at making it easier to clean up the annotations we have. The STAC sanity checker can be used to see what problems there are with the current crop of annotations.

The sanity checker is best run in easy mode in the STAC development directory (ie. the project SVN at the time of this writing):

```
stac-check --doc pilot03
```

It will output a report directory in a temporary location (something like */tmp/sanity-pilot03/*). The report will be in HTML (with links to some styled XML documents and SVG graphs) and so should be viewed in a browser.

1.1.3 stac-edit and stac-oneoff

stac-edit and stac-oneoff are probably best reserved for people interested in refining the annotations in the STAC corpus. See the *–help* options for these tools or get in touch with us for our internal documentation

1.1.4 User interface notes

Command line filters

The stac utilities tend to use the same idiom of filtering the corpus on the command line. For example, the following command will try to display the text for all (sub)documents in the training-2015-05-30 corpus whose document names start with “pilot”; and subdocument is either ‘02’, ‘03’, or ‘04’; and which in the ‘discourse’ stage and by the annotator ‘GOLD’

```
stac-util text --doc 'pilot'\
               --subdoc '0[2-4]'\
               --stage 'discourse'\
               --anno 'GOLD'\
    data/FROZEN/training-2015-05-30
```

As we can see above, the filters are Python regular expressions, which can sometimes be useful for expressing range matches. It's also possible to filter as much or as little as you want, for example with this subcommand showing EVERY gold-annotated document in that corpus

```
stac-util text --anno 'GOLD' data/FROZEN/training-2015-05-30
```

Or this command which displays every single document there is

```
stac-util text data/FROZEN/training-2015-05-30
```

Easy mode

The commands generally come with an "easy mode" where you need only specify a single document via '–doc'

```
stac-util text --doc pilot03
```

If you do this, the stac utilities will guess that you wanted the development corpus directory and sometimes some sensible flags to go with it.

Note that "easy mode" does not preclude the use of other flags; you could also still have complex filters like the following

```
stac-util text --doc pilot03 --subdoc '0[2-4]' --anno GOLD
```

Easy mode is available for stac-check, stac-edit, stac-oneoff, and stac-util.

CHAPTER 2

Tutorial

Note: if you have downloaded the educe source code, the tutorial is available as iPython notebooks in the doc directory

2.1 STAC

Educe is a library for working with a variety of discourse corpora. This tutorial aims to show what using educe would be like when working with the STAC corpus.

We'll be working with a tiny fragment of the corpus included with educe. You may find it useful to symlink your larger copy from the STAC distribution and modify this tutorial accordingly.

2.1.1 Installation

```
git clone https://github.com/irit-melodi/educe.git
cd educe
pip install -r requirements.txt
```

Note: these instructions assume you are running within a virtual environment. If not, and if you have permission denied errors, replace `pip` with `sudo pip`.

2.1.2 Tutorial in browser (optional)

This tutorial can either be followed along with the command line and your favourite text editor, or embedded in an interactive webpage via iPython:

```
pip install ipython
cd tutorials
ipython notebook
```

```
# some helper functions for the tutorial below

def text_snippet(text):
    "short text fragment"
    if len(text) < 43:
        return text
    else:
        return "{0}...{1}".format(text[:20], text[-20:])

def highlight(astring, color=1):
    "coloured text"
    return("\x1b[3{color}m{str}\x1b[0m".format(color=color, str=astring))
```

2.1.3 Reading corpus files (STAC)

Typically, the first thing we want to do when working in educe is to read the corpus in. This can be a bit slow, but as we will see later on, we can speed things up if we know what we're looking for.

```
from __future__ import print_function
import educe.stac

# relative to the educe docs directory
data_dir = '../data'
corpus_dir = '{dd}/stac-sample'.format(dd=data_dir)

# read everything from our sample
reader = educe.stac.Reader(corpus_dir)
corpus = reader.slurp(verbose=True)

# print a text fragment from the first ten files we read
for key in corpus.keys()[:10]:
    doc = corpus[key]
    print("[{0}] {1}".format(key, doc.text()[:50]))
```

```
Slurping corpus dir [99/100]
```

```
[s1-league2-game1 [05] unannotated None]  199 : sabercat : anyone any clay? 200 : IG␣
↪: nope
[s1-league2-game1 [13] units hjoseph]  521 : sabercat : skinnylinny 522 : sabercat :␣
↪som
[s1-league2-game1 [10] units hjoseph]  393 : skinnylinny : Shall we extend? 394 :␣
↪saberc
[s1-league2-game1 [11] discourse hjoseph]  450 : skinnylinny : Argh 451 : skinnylinny␣
↪: How
[s1-league2-game1 [10] unannotated None]  393 : skinnylinny : Shall we extend? 394 :␣
↪saberc
[s1-league2-game1 [02] units lpetersen]  75 : sabercat : anyone has any wood? 76 :␣
↪skinnyl
[s1-league2-game1 [14] units SILVER]  577 : sabercat : skinny 578 : sabercat : I need␣
↪2
[s1-league2-game3 [03] discourse lpetersen]  151 : amycharl : got wood anyone? 152 :␣
↪sabercat
[s1-league2-game1 [10] discourse hjoseph]  393 : skinnylinny : Shall we extend? 394 :␣
↪saberc
[s1-league2-game1 [12] units SILVER]  496 : sabercat : yes! 497 : sabercat : :D 498 :␣
↪s
```

```
Slurping corpus dir [100/100 done]
```

Faster reading

If you know that you only want to work with a subset of the corpus files, you can pre-filter the corpus before reading the files.

It helps to know here that an educe corpus is a mapping from file id keys to Documents. The `FileId` tells us what makes a Document distinct from another:

- document (eg. s1-league2-game1): in STAC, the game that was played (here, season 1, league 2, game 1)
- subdocument (eg. 05): a mostly arbitrary subdivision of the documents motivated by technical constraints (overly large documents would cause our annotation tool to crash)
- stage (eg. units, discourse, parsed): the kinds of annotations available in the document
- annotator (eg. hjoseph): the main annotator for a document (gold standard documents have the distinguished annotators, BRONZE, SILVER, or GOLD)

NB: unfortunately we have overloaded the word "document" here. When talking about file ids, "document" refers to a whole game. But when talking about actual annotation objects an educe Document actually corresponds to a specific combination of document, subdocument, stage, and annotator

```
import re

# nb: you can import this function from educe.stac.corpus
def is_metal(fileid):
    "is this a gold standard(ish) annotation file?"
    anno = fileid.annotator or ""
    return anno.lower() in ["bronze", "silver", "gold"]

# pick out gold-standard documents
subset = reader.filter(reader.files(),
                       lambda k: is_metal(k) and int(k.subdoc) < 4)
corpus_subset = reader.slurp(subset, verbose=True)
for key in corpus_subset:
    doc = corpus_subset[key]
    print("{0}: {1}".format(key, doc.text()[:50]))
```

```
Slurping corpus dir [11/12]
```

```
s1-league2-game1 [01] units SILVER:  1 : sabercat : btw, are we playing without the ot
s1-league2-game1 [01] discourse SILVER:  1 : sabercat : btw, are we playing without␣
↪the ot
s1-league2-game1 [02] discourse SILVER:  75 : sabercat : anyone has any wood? 76 :␣
↪skinnyl
s1-league2-game3 [01] discourse BRONZE:  1 : amycharl : i made it! 2 : amycharl : did␣
↪the
s1-league2-game1 [03] discourse SILVER:  109 : sabercat : well done! 110 : IG : More␣
↪clay!
s1-league2-game3 [02] units BRONZE:  73 : sabercat : skinny, got some ore? 74 : skinny
s1-league2-game3 [01] units BRONZE:  1 : amycharl : i made it! 2 : amycharl : did the
s1-league2-game1 [02] units SILVER:  75 : sabercat : anyone has any wood? 76 : skinnyl
s1-league2-game3 [02] discourse BRONZE:  73 : sabercat : skinny, got some ore? 74 :␣
↪skinny
```

```
s1-league2-game1 [03] units SILVER:  109 : sabercat : well done! 110 : IG : More clay!
s1-league2-game3 [03] discourse BRONZE:  151 : amycharl : got wood anyone? 152 :␣
↪sabercat
s1-league2-game3 [03] units BRONZE:  151 : amycharl : got wood anyone? 152 : sabercat
```

```
Slurping corpus dir [12/12 done]
```

```
from educe.corpus import FileId

# pick out an example document to work with creating FileIds by hand
# is not something we would typically do (normally we would just iterate
# through a corpus), but it's useful for illustration
ex_key = FileId(doc='s1-league2-game3',
                subdoc='03',
                stage='units',
                annotator='BRONZE')
ex_doc = corpus[ex_key]
print(ex_key)
```

```
s1-league2-game3 [03] units BRONZE
```

2.1.4 Standing off

Most annotations in the STAC corpus are educe standoff annotations. In educe terms, this means that they (perhaps indirectly) extend the `educe.annotation.Standoff` class and provide a `text_span()` function. Much of our reasoning around annotations essentially consists of checking that their text spans overlap or enclose each other.

As for the text spans, these refer to the raw text saved in files with an `.ac` extension (eg. `s1-league1-game3.ac`). In the Glozz annotation tool, these `.ac` text files form a pair with their `.aa` xml counterparts. Multiple annotation files can point to the same text file.

There are also some annotations that come from 3rd party tools, which we will uncover later.

2.1.5 Documents and EDUs

A document is a sort of giant annotation that contains three other kinds of annotation

- units - annotations that directly cover a span of text (EDUs, Resources, but also turns, dialogues)
- relations - annotations that point from one annotation to another
- schemas - annotations that point to a set of annotations

To start things off, we'll focus on one type of unit-level annotation, the Elementary Discourse Unit

```
def preview_unit(doc, anno):
    "the default str(anno) can be a bit overwhelming"
    preview = "{span: <11} {id: <20} [{type: <12}] {text}"
    text = doc.text(anno.text_span())
    return preview.format(id=anno.local_id(),
                          type=anno.type,
                          span=anno.text_span(),
                          text=text_snippet(text))

print("Example units")
print("-------------")
```

```
seen = set()
for anno in ex_doc.units:
    if anno.type not in seen:
        seen.add(anno.type)
        print(preview_unit(ex_doc, anno))

print()
print("First few EDUs")
print("--------------")
for anno in filter(educe.stac.is_edu, ex_doc.units)[:4]:
    print(preview_unit(ex_doc, anno))
```

```
Example units
-------------
(1,34)      stac_1368693094      [paragraph   ] 151 : amycharl : got wood anyone?
(52,66)     stac_1368693099      [Accept      ] yep, for what?
(117,123)   stac_1368693105      [Refusal     ] no way
(189,191)   stac_1368693114      [Other       ] :)
(209,210)   stac_1368693117      [Counteroffer] ?
(659,668)   stac_1368693162      [Offer       ] how much?
(22,26)     asoubeille_1374939590843 [Resource    ] wood
(35,66)     stac_1368693098      [Turn        ] 152 : sabercat : yep, for what?
(0,266)     stac_1368693124      [Dialogue    ]  151 : amycharl : go...cat : yep,␣
↪thank you

First few EDUs
--------------
(52,66)     stac_1368693099      [Accept      ] yep, for what?
(117,123)   stac_1368693105      [Refusal     ] no way
(163,171)   stac_1368693111      [Accept      ] could be
(189,191)   stac_1368693114      [Other       ] :)
```

2.1.6 TODO

Everything below this point should be considered to be in a scratch/broken state. It needs to ported over from its RST/DT considerations to STAC

To do:

- standing off (ac/aa) - shared aa
- layers (units/discourse)
- working with relations and schemas
- grabbing resources etc (example of working with unit level annotation)
- synchronising layers (grabbing the dialogue act and relations at the same time)
- external annotations (postags, parse trees)
- working with hypergraphs (implementing `_repr_png()_` would be pretty sweet)

Tree searching

The same span enclosure logic can be used to search parse trees for particular constituents, verb phrases. Alternatively, you can use the the `topdown` method provided by educe trees. This returns just the largest constituent for which some predicate is true. It optionally accepts an additional argument to cut off the search when it is clearly out of bounds.

2.1.7 Conclusion

In this tutorial, we've explored a couple of basic educe concepts, which we hope will enable you to extract some data from your discourse corpora, namely

- reading corpus data (and pre-filtering)
- standoff annotations
- searching by span enclosure, overlapping
- working with trees
- combining annotations from different sources

The concepts above should transfer to whatever discourse corpus you are working with (that educe supports, or that you are prepared to supply a reader for).

Work in progress

This tutorial is very much a work in progress (last update: 2014-09-19). Educe is a bit of a moving target, so let me know if you run into any trouble!

See also

stac-util

Some of the things you may want to do with the STAC corpus may already exist in the stac-util command line tool. stac-util is meant to be a sort of Swiss Army Knife, providing tools for editing the corpus. The query tools are more likely to be of interest:

- text: display text and edu/dialogue segmentation in a friendly way
- graph: draw discourse graphs with graphviz (arrows for relations, boxes for CDUs, etc)
- filter-graph: visualise instances of relations (eg. Question answer pair)
- count: generate statistics about the corpus

See `stac-util --help` for more details.

External tool support

Educe has some support for reading data from outside the discourse corpus proper. For example, if you run the stanford corenlp parser on the raw text, you can read them back into educe-style `ConstituencyTree` and `DependencyTree` annotations. See educe.external for details.

If you have a part of speech tagger that you would like to use, the `educe.external.postag` module may be useful for representing the annotations that come out of it

You can also add support for your own tools by creating annotations that extend `Standoff`, directly or otherwise.

2.2 RST-DT

Educe is a library for working with a variety of discourse corpora. This tutorial aims to show what using educe would be like.

2.2.1 Installation

```
git clone https://github.com/irit-melodi/educe.git
cd educe
pip install -r requirements.txt
```

Note: these instructions assume you are running within a virtual environment. If not, and if you have permission denied errors, replace `pip` with `sudo pip`.

2.2.2 Tutorial setup

RST-DT portions of this tutorial require that you have a local copy of the RST Discourse Treebank. For purposes of this tutorial, you will need to link this into the data directory, for example

```
ln -s $HOME/CORPORA/rst_discourse_treebank data
ln -s $HOME/CORPORA/PTBIII data
```

Tutorial in browser (optional)

This tutorial can either be followed along with the command line and your favourite text editor, or embedded in an interactive webpage via iPython:

```
pip install ipython
cd tutorials
ipython notebook
```

2.2.3 Reading corpus files (RST-DT)

```
from __future__ import print_function
import educe.rst_dt

# relative to the educe docs directory
data_dir = '../data'
rst_corpus_dir = '{dd}/rst_discourse_treebank/data/RSTtrees-WSJ-double-1.0/'.
↪format(dd=data_dir)

# read and load the documents from the WSJ which were double-tagged
rst_reader = educe.rst_dt.Reader(rst_corpus_dir)
rst_corpus = rst_reader.slurp(verbose=True)

# print a text fragment from the first ten files we read
for key in rst_corpus.keys()[:10]:
    doc = rst_corpus[key]
    print("{0}: {1}".format(key.doc, doc.text()[:50]))
```

```
Slurping corpus dir [51/53]
```

```
wsj_1365.out: The Justice Department has revised certain interna
wsj_0633.out: These are the last words Abbie Hoffman ever uttere
wsj_1105.out: CHICAGO - Sears, Roebuck & Co. is struggling as it
wsj_1168.out: Wang Laboratories Inc. has sold $25 million of ass
wsj_1100.out: Westinghouse Electric Corp. said it will buy Shaw-
wsj_1924.out: CALIFORNIA STRUGGLED with the aftermath of a Bay a
wsj_0669.out: Nissan Motor Co. expects net income to reach 120 b
wsj_0651.out: Nelson Holdings International Ltd. shareholders ap
wsj_2309.out: Atco Ltd. said its utilities arm is considering bu
wsj_1120.out: Japan has climbed up from the ashes of World War I
```

```
Slurping corpus dir [53/53 done]
```

Faster reading

If you know that you only want to work with a subset of the corpus files, you can pre-filter the corpus before reading the files.

It helps to know here that an educe corpus is a mapping from file id keys to documents. The FileId contains the minimally identifying metadata for a document, for example, the document name, or its annotator. For RST-DT, only the `doc` attribute is used.

```
rst_subset = rst_reader.filter(rst_reader.files(),
                               lambda k:k.doc.startswith("wsj_062"))
rst_corpus_subset = rst_reader.slurp(rst_subset, verbose=True)
for key in rst_corpus_subset:
    doc = rst_corpus_subset[key]
    print("{0}: {1}".format(key.doc, doc.text()[:50]))
```

```
wsj_0627.out: October employment data -- also could turn out to
wsj_0624.out: Costa Rica reached an agreement with its creditor
```

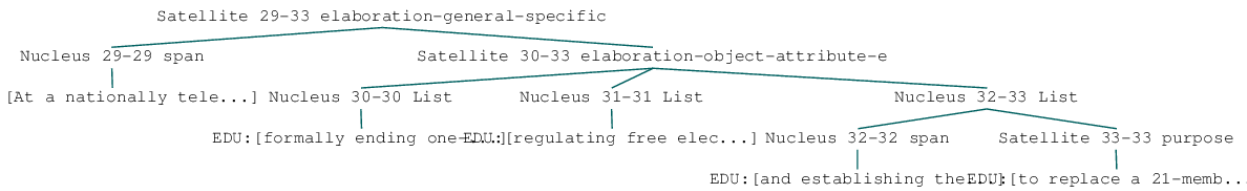
```
Slurping corpus dir [2/2 done]
```

2.2.4 Trees and annotations

RST DT documents are basically trees

```
from educe.corpus import FileId
# an (ex)ample document
ex_key = educe.rst_dt.mk_key("wsj_1924.out")
ex_doc = rst_corpus[ex_key] # pick a document from the corpus

# display PNG tree
from IPython.display import display
ex_subtree = ex_doc[2][0][0][1] # navigate down to a small subtree
display(ex_subtree) # NLTK > 3.0b1 2013-07-11 should display a PNG image of the RST␣
↪tree
# Mac users: see note below
```



Note for Mac users following along in iPython: if displaying the tree above does not work (particularly if you see a GS prompt in your iPython terminal window instead of an embedded PNG in your browser), try my NLTK patch from 2014-09-17.

Standing off

RST DT trees function both as NLTK trees, and as educe standoff annotations. Most annotations in educe can be seen as standoff annotations in some sense; they (perhaps indirectly) extend `educe.annotation.Standoff` and provide a `text_span()` function. Comparing annotations usually consists of comparing their text spans.

Text spans in the RST DT corpus refer to the source document beneath each tree file, eg. for the tree file `wsj_1111.out.dis`, educe reads `wsj_1111.out` as its source text. (The source text is somewhat optional as the RST trees themselves contain text, but this tends to have subtle differences with its underlying source). Below, we see an example of one of these source documents.

```
ex_rst_txt_filename = '{corpus}/{doc}'.format(corpus=rst_corpus_dir,
                                              doc=ex_key.doc)

with open(ex_rst_txt_filename) as ifile:
    ex_txt = ifile.read()
    ex_snippet_start = ex_txt.find("At a national")
    print(ex_txt[ex_snippet_start:ex_snippet_start + 500])
```

```
At a nationally televised legislative session in Budapest, the Parliament␣
↪overwhelmingly approved changes formally ending one-party domination in the country,
↪ regulating free elections by next summer and establishing the office of state␣
↪president to replace a 21-member council.
The country was renamed the Republic of Hungary.
Like other Soviet bloc nations, it had been known as a "people's republic" since

The voting for new laws followed dissolution of Hungary's Communist Party this month␣
↪and
```

Now let's have a closer look at the annotations themselves.

```
# it may be useful to have a couple of helper functions to
# display standoff annotations in a generic way
def text_snippet(text):
    "short text fragment"
    if len(text) < 43:
        return text
    else:
        return "{0}...{1}".format(text[:20], text[-20:])

def preview_standoff(tystr, context, anno):
    "simple glimpse at a standoff annotation"
    span = anno.text_span()
    text = context.text(span)
    return "{tystr} at {span}:\t{snippet}".format(tystr=tystr,
```

```
                                                    span=span,
                                                    snippet=text_snippet(text))
```

EDUs and subtrees

```
# in educe RST/DT all annotations have a shared context object
# that refers to an RST document; you don't always need to use
# it, but it can be handy for writing general code like the
# above
ex_context = ex_doc.label().context

# display some edus
print("Some edus")
edus = ex_subtree.leaves()
for edu in edus:
    print(preview_standoff("EDU", ex_context, edu))

print("\nSome subtrees")
# display some RST subtrees and the edus they enclose
for subtree in ex_subtree.subtrees():
    node = subtree.label()
    stat = "N" if node.is_nucleus() else "S"
    label = "{stat} {rel: <30}".format(stat=stat,
                                       rel=node.rel)
    print(preview_standoff(label, ex_context, subtree))
```

```
Some edus
EDU at (1504,1609): At a nationally tele...gly approved changes
EDU at (1610,1662): formally ending one-...tion in the country,
EDU at (1663,1703): regulating free elections by next summer
EDU at (1704,1750): and establishing the...e of state president
EDU at (1751,1782): to replace a 21-member council.

Some subtrees
S elaboration-general-specific   at (1504,1782):    At a nationally tele...a 21-
↪member council.
N span                           at (1504,1609):    At a nationally tele...gly␣
↪approved changes
S elaboration-object-attribute-e at (1610,1782):    formally ending one-...a 21-
↪member council.
N List                           at (1610,1662):    formally ending one-...tion in␣
↪the country,
N List                           at (1663,1703):    regulating free elections by next␣
↪summer
N List                           at (1704,1782):    and establishing the...a 21-
↪member council.
N span                           at (1704,1750):    and establishing the...e of state␣
↪president
S purpose                        at (1751,1782):    to replace a 21-member council.
```

Paragraphs and sentences

Going back to the source text, we can notice that it seems to be divided into sentences and paragraphs with line separators. This does not seem to be done very consistently, and in any case, RST constituents seem to traverse these

boundaries freely. But they can still make for useful standoff annotations.

```
for para in ex_context.paragraphs[4:8]:
    print(preview_standoff("paragraph", ex_context, para))
    for sent in para.sentences:
        print("\t" + preview_standoff("sentence", ex_context, sent))
```

```
paragraph at (862,1288):    The 77-year-old offi...o-democracy groups.
    sentence at (862,1029): The 77-year-old offi...ttee in East Berlin.
    sentence at (1030,1144):        Honecker, who was re... for health reasons.
    sentence at (1145,1288):        He was succeeded by ...o-democracy groups.
paragraph at (1290,1432):   Honecker's departure...nted with his rule.
    sentence at (1290,1432):        Honecker's departure...nted with his rule.
paragraph at (1434,1502):   HUNGARY ADOPTED cons... democratic system.
    sentence at (1434,1502):        HUNGARY ADOPTED cons... democratic system.
paragraph at (1504,1913):   At a nationally tele...e's republic" since
    sentence at (1504,1782):        At a nationally tele...a 21-member council.
    sentence at (1783,1831):        The country was rena...Republic of Hungary.
    sentence at (1832,1913):        Like other Soviet bl...e's republic" since
```

2.2.5 Penn Treebank integration

RST DT annotations are mostly over Wall Street Journal articles from the Penn Treebank. If you have a copy of the latter at the ready, you can ask educe to read and align the two (ie. PTB annotations treated as standing off the RST source text). This alignment consists of some universal substitutions (eg. `-LBR-` to `()` and with a bit of hardcoding to account for seemingly random differences in whitespace/punctuation.

```
from educe.rst_dt import ptb
from nltk.tree import Tree

# confusingly, this is not an educe corpus reader, but the NLTK
# bracketed reader.  Sorry
ptb_reader = ptb.reader('{dd}/PTBIII/parsed/mrg/wsj/'.format(dd=data_dir))
ptb_trees = {}
for key in rst_corpus:
    ptb_trees[key] = ptb.parse_trees(rst_corpus, key, ptb_reader)

# pick and display an arbitary ptb tree
ex0_ptb_tree = ptb_trees[rst_corpus.keys()[0]][0]
print(ex0_ptb_tree.pprint()[:400])
```

```
(S
  (NP-SBJ
    (DT <educe.external.postag.Token object at 0x10e41ecd0>)
    (NNP <educe.external.postag.Token object at 0x10e41ee10>)
    (NNP <educe.external.postag.Token object at 0x10e41ef50>))
  (VP
    (VBZ <educe.external.postag.Token object at 0x10e41efd0>)
    (VP
      (VP
        (VBN <educe.external.postag.Token object at 0x10e41ef90>)
        (NP
          (JJ <educe.external.postag.
```

The result of this alignment is an educe `ConstituencyTree`, the leaves of which are educe `Token` objects. We'll say a little bit more about these below.

```
# show what's beneath these educe tokens
def str_tree(tree):
    if isinstance(tree, Tree):
        return Tree(str(tree.label()), map(str_tree, tree))
    else:
        return str(tree)

print(str_tree(ex0_ptb_tree).pprint()[:400])
```

```
(S
  (NP-SBJ
    (DT The/DT      (0,3))
    (NNP Justice/NNP        (4,11))
    (NNP Department/NNP     (12,22)))
  (VP
    (VBZ has/VBZ    (23,26))
    (VP
      (VP
        (VBN revised/VBN    (27,34))
        (NP
          (JJ certain/JJ    (35,42))
          (JJ internal/JJ   (43,51))
          (NNS guidelines/NNS       (52,62))))
      (CC and/CC    (63,66))
      (VP (VBN clarified/VBN        (67,76)) (NP (NNS others/NNS    (77,83))))
```

2.2.6 Combining annotations

We now have several types of annotation at our disposal:

- EDUs and RST trees
- raw text paragraph/sentences (not terribly reliable)
- PTB trees

The next question that arises is how we can use these annotations in conjuction with each other.

Span enclosure and overlapping

The simplest way to reason about annotations (particularly since they tend to be sloppy and to overlap). Suppose for example, we wanted to find all of the edus in a tree that are in the same sentence as an given edu.

```
from itertools import chain

# pick an EDU, any edu
ex_edus = ex_subtree.leaves()
ex_edu0 = ex_edus[3]
print(preview_standoff('example EDU', ex_context, ex_edu0))

# all of the sentences in the example document
ex_sents = list(chain.from_iterable(x.sentences for x in ex_context.paragraphs))

# sentences that overlap the edu
# (we use overlaps instead of encloses because edus might
# span sentence boundaries)
```

```
ex_edu0_sents = [x for x in ex_sents if x.overlaps(ex_edu0)]

# and now the edus that overlap those sentences
ex_edu0_buddies = []
for sent in ex_edu0_sents:
    print(preview_standoff('overlapping sentence', ex_context, sent))
    buddies = [x for x in ex_edus if x.overlaps(sent)]
    buddies.remove(ex_edu0)
    for edu in buddies:
        print(preview_standoff('\tnearby EDU', ex_context, edu))
    ex_edu0_buddies.extend(buddies)
```

```
example EDU at (1704,1750): and establishing the...e of state president
overlapping sentence at (1504,1782):        At a nationally tele...a 21-member␣
↪council.
    nearby EDU at (1504,1609):      At a nationally tele...gly approved changes
    nearby EDU at (1610,1662):      formally ending one-...tion in the country,
    nearby EDU at (1663,1703):      regulating free elections by next summer
    nearby EDU at (1751,1782):      to replace a 21-member council.
```

Span example 2 (exercise)

As an exercise, how about extracting the PTB part of speech tags for every token in our example EDU? How for example, would you determine if an EDU contains a VBG-tagged word?

```
ex_postags = list(chain.from_iterable(t.leaves() for t in ptb_trees[ex_key]))

print("some of the POS tags")
for postag in ex_postags[300:310]:
    print(preview_standoff(postag.tag, ex_context, postag))

print()
ex_edu0_postags = [] # EXERCISE <-- fill this in
print("has VBG? ", ) # EXERCISE <-- fill this in
```

```
some of the POS tags
VBG at (1663,1673): regulating
JJ at (1674,1678):  free
NNS at (1679,1688): elections
IN at (1689,1691):  by
JJ at (1692,1696):  next
NN at (1697,1703):  summer
CC at (1704,1707):  and
VBG at (1708,1720): establishing
DT at (1721,1724):  the
NN at (1725,1731):  office

has VBG?
```

Tree searching

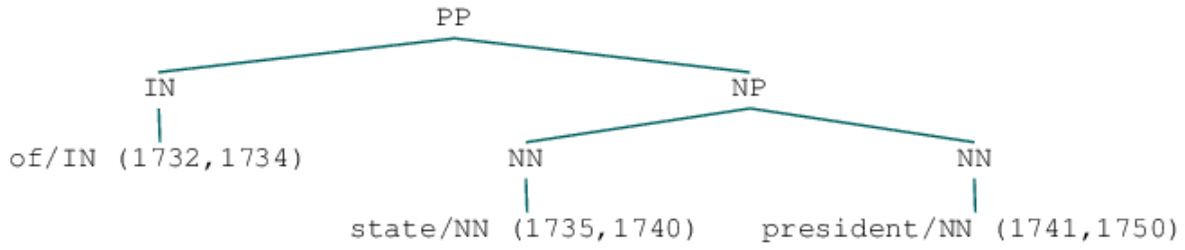
The same span enclosure logic can be used to search parse trees for particular constituents, verb phrases. Alternatively, you can use the the `topdown` method provided by educe trees. This returns just the largest constituent for which some predicate is true. It optionally accepts an additional argument to cut off the search when it is clearly out of bounds.

```
ex_ptb_trees = ptb_trees[ex_key]
ex_edu0_ptb_trees = [x for x in ex_ptb_trees if x.overlaps(ex_edu0)]
ex_edu0_cons = []
for ptree in ex_edu0_ptb_trees:
    print(preview_standoff('ptb tree', ex_context, ptree))
    ex_edu0_cons.extend(ptree.topdown(lambda c: ex_edu0.encloses(c)))

# the largest constituents enclosed by this edu
for cons in ex_edu0_cons:
    print(preview_standoff(cons.label(), ex_context, cons))

display(ex_edu0_cons[3])
```

```
ptb tree at (1504,1782):    At a nationally tele...a 21-member council.
CC at (1704,1707):  and
VBG at (1708,1720): establishing
NP at (1721,1731):  the office
PP at (1732,1750):  of state president
WHNP-1 at (1750,1750):
NP-SBJ at (1750,1750):
```

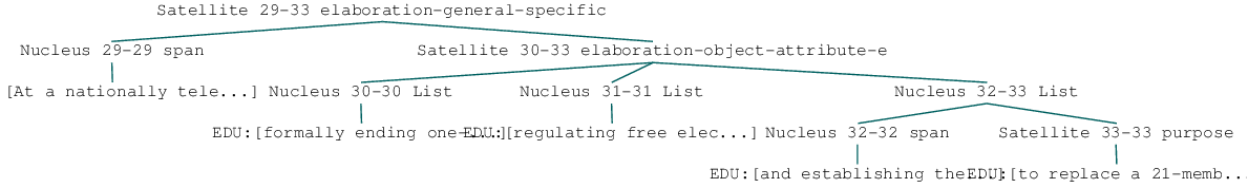


2.2.7 Simplified trees

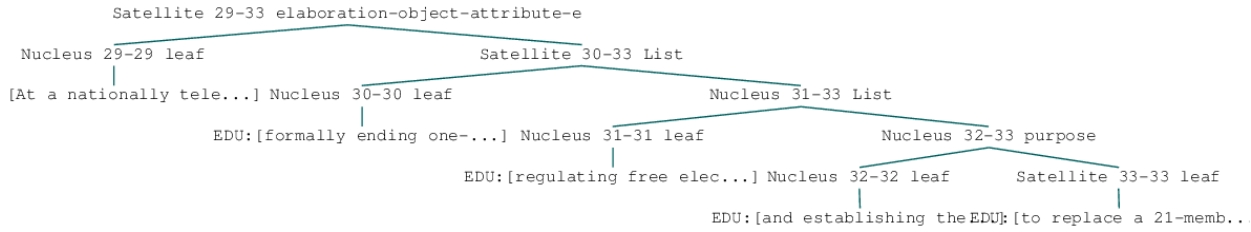
The tree representation used in the RST DT can take some getting used to (relation labels are placed on the satellite rather than the root of a subtree). You may prefer to work with the simplified representation instead. In the simple representation, trees are binarised and relation labels are moved to the root node. Compare for example, the two versions of the same RST subtree.

```
# rearrange the tree so that it is easier to work with
ex_simple_subtree = educe.rst_dt.SimpleRSTTree.from_rst_tree(ex_subtree)
print('Corpus representation\n\n')
display(ex_subtree)
print('Simplified (binarised, rotated) representation\n\n')
display(ex_simple_subtree)
```

```
Corpus representation
```



```
Simplified (binarised, rotated) representation
```



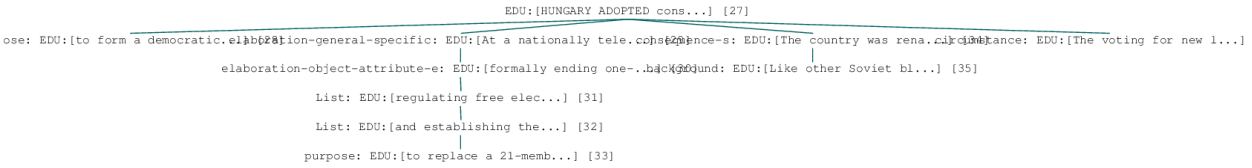
2.2.8 Dependency trees and back

Educe also provides an experimental conversion between simplified trees above and dependency trees. See the educe.rst_dt.deptree for the algorithm used.

Our current example is a little too small to give a sense of what the resulting dependency tree might look like, so we'll back up slightly closer to the root to have a wider view.

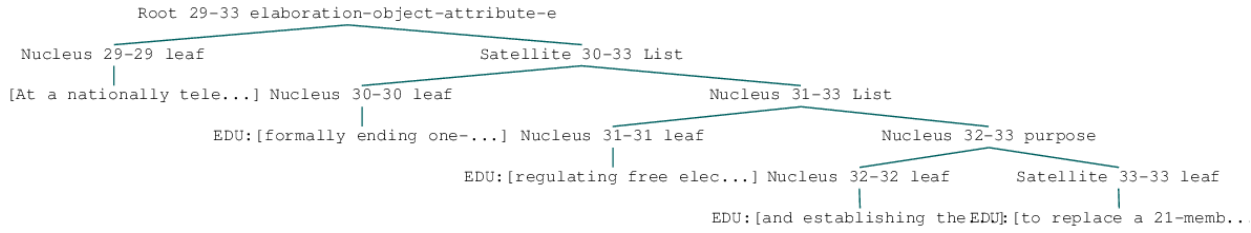
```
from educe.rst_dt import deptree

ex_subtree2 = ex_doc[2]
ex_simple_subtree2 = educe.rst_dt.SimpleRSTTree.from_rst_tree(ex_subtree2)
ex_deptree2 = deptree.relaxed_nuclearity_to_deptree(ex_simple_subtree2)
display(ex_deptree2)
```



Going back to our original example, we can (lossily) convert back from these dependency tree representations to RST trees. The dependency trees have some ambiguities in them that we can't resolve without an oracle, but we can at least make some guesses. Note that when converting back to RST, we need to supply a list of relation labels that should be treated as multinuclear.

```
ex_deptree = deptree.relaxed_nuclearity_to_deptree(ex_simple_subtree)
ex_from_deptree = deptree.relaxed_nuclearity_from_deptree(ex_deptree, ["list"]) #␣
↪multinuclear in lowercase
display(ex_from_deptree)
```



2.2.9 Conclusion

In this tutorial, we've explored a couple of basic educe concepts, which we hope will enable you to extract some data from your discourse corpora, namely

- reading corpus data (and pre-filtering)
- standoff annotations
- searching by span enclosure, overlapping
- working with trees
- combining annotations from different sources

The concepts above should transfer to whatever discourse corpus you are working with (that educe supports, or that you are prepared to supply a reader for).

That said, some of the features mentioned in particular tutorial are specific to the RST DT:

- simplifying RST trees
- converting them to dependency trees
- PTB integration

This tutorial was last updated on 2014-09-18. Educe is a bit of a moving target, so let me know if you run into any trouble!

See also

rst-dt-util

Some of the things you may want to do with the RST DT may already exist in the rst-dt-util command line tool. See `rst-dt-util --help` for more details.

(At the time of this writing the only really useful tool is the rst-dt-util reltypes one, which prints an inventory of relation labels, but the utility may grow over time)

External tool support

Educe has some support for reading data from outside the discourse corpus proper. For example, if you run the stanford corenlp parser on the raw text, you can read them back into educe-style `ConstituencyTree` and `DependencyTree` annotations. See educe.external for details.

If you have a part of speech tagger that you would like to use, the `educe.external.postag` module may be useful for representing the annotations that come out of it

You can also add support for your own tools by creating annotations that extend `Standoff`, directly or otherwise.

2.3 PDTB

Educe is a library for working with a variety of discourse corpora. This tutorial aims to show what using educe would be like when working with the Penn Discourse Treebank corpus.

2.3.1 Installation

```
git clone https://github.com/kowey/educe.git
cd educe
pip install -r requirements.txt
```

Note: these instructions assume you are running within a virtual environment. If not, and if you have permission denied errors, replace `pip` with `sudo pip`.

2.3.2 Tutorial setup

This tutorial require that you have a local copy of the PDTB. For purposes of this tutorial, you will need to link this into the data directory, for example

```
ln -s $HOME/CORPORA/pdtb_v2 data
```

Optionnally, to match the pdtb text spans to their analysis in the Penn Treebank, you need to have a local copy of the PTB at the same location

```
ln -s $HOME/CORPORA/PTBIII data
```

Tutorial in browser (optional)

This tutorial can either be followed along with the command line and your favourite text editor, or embedded in an interactive webpage via iPython:

```
pip install ipython
cd tutorials
ipython notebook
```

```
# some helper functions for the tutorial below

def show_type(rel):
    "short string for a relation type"
    return type(rel).__name__[:-8]  # remove "Relation"

def highlight(astring, color=1):
    "coloured text"
    return("\x1b[3{color}m{str}\x1b[0m".format(color=color, str=astring))
```

2.3.3 Reading corpus files (PDTB)

NB: unfortunately, at the time of this writing, PDTB support in educe is very much behind and rather inconsistent with that of the other corpora. Apologies for the mess!

```
from __future__ import print_function
import educe.pdtb

# relative to the educe docs directory
data_dir = '../data'
corpus_dir = '{dd}/pdtb_v2/data'.format(dd=data_dir)

# read a small sample of the pdtb
reader = educe.pdtb.Reader(corpus_dir)
anno_files = reader.filter(reader.files(),
                           lambda k: k.doc.startswith('wsj_231'))
corpus = reader.slurp(anno_files, verbose=True)

# print the first five rel types we read from each doc
```

```
for key in corpus.keys()[:10]:
    doc = corpus[key]
    rtypes = [show_type(r) for r in doc]
    print("[{0}] {1}".format(key.doc, " ".join(rtypes[:5])))
```

```
Slurping corpus dir [7/8]
```

```
[wsj_2315] Explicit Implicit Entity Explicit Implicit
[wsj_2311] Implicit
[wsj_2316] Explicit Implicit Implicit Implicit Explicit
[wsj_2310] Entity
[wsj_2319] Explicit
[wsj_2317] Implicit Implicit Explicit Implicit Explicit
[wsj_2313] Entity Explicit Explicit Implicit Explicit
[wsj_2314] Explicit Explicit Implicit Explicit Entity
```

```
Slurping corpus dir [8/8 done]
```

2.3.4 What's a corpus?

A corpus is a dictionary from `FileId` keys to representation of PDTB documents.

Keys

A key has several fields meant to distinguish different annotated documents from each other. In the case of the PDTB, the only field of interest is `doc`, a Wall Street journal article number as you might find in the PTB.

```
ex_key = educe.pdtb.mk_key('wsj_2314')
ex_doc = corpus[ex_key]

print(ex_key)
print(ex_key.__dict__)
```

```
wsj_2314 [None] discourse unknown
{'doc': 'wsj_2314', 'subdoc': None, 'annotator': 'unknown', 'stage': 'discourse'}
```

Documents

At some point in the future, the representation of a document may change to something a bit higher level and easier to work with. For now, a "document" in the educe PDTB sense consists of a list of relations, each relation having a low-level representation that hews fairly closely to the grammar described in the PDTB annotation manual.

TIP: At least until educe grows a more educe-like uniform representation of PDTB annotations, a very useful resource to look at when working with the PDTB may be The Penn Discourse Treebank 2.0 Annotation Manual, sections 6.3.1 to 6.3.5 (Description of PDTB representation format → File format → General outline...).

```
lr = [r for r in ex_doc]
r0 = lr[0]
type(r0).__name__
```

```
'ExplicitRelation'
```

Relations

There are five types of relation annotation: explicit, implicit, altlex, entity, no (as in no relation). These are described in further detail in the PDTB annotation manual. Here's well try to sketch out some of the important properties.

The main thing to notice is that the 5 types of annotation not have very much in common with each other, but they have many overlapping pieces (see table in the educe.pdtb docs)

- a relation instance always has two arguments (these can be selected as `arg1` and `arg2`)

```
def display_rel(r):
    "pretty print a relation instance"

    rtype = show_type(r)

    if rtype == "Explicit":
        conn = highlight(r.connhead)
    elif rtype == "Implicit":
        conn = "{rtype} {conn1}".format(rtype=rtype,
                                        conn1=highlight(str(r.connective1)))
    elif rtype == "AltLex":
        conn = "{rtype} {sem1}".format(rtype=rtype,
                                       sem1=highlight(r.semclass1))
    else:
        conn = rtype

    fmt = "{src}\n \t ---[{label}]---->\n \t\t\t{tgt}"
    return(fmt.format(src=highlight(r.arg1.text, 2),
                      label=conn,
                      tgt=highlight(r.arg2.text, 2)))


print(display_rel(r0))
```

```
[32mQuantum Chemical Corp. went along for the ride[0m
     ---[[31mConnective(when | Temporal.Synchrony)[0m]---->
                     [32mthe price of plastics took off in 1987[0m
```

```
r0.connhead.text
```

```
u'when'
```

2.3.5 Gorn addresses

```
# print the first seven gorn addresses for the first argument of the first
# 5 rels we read from each doc
for key in corpus.keys()[:3]:
    doc = corpus[key]
    rels = doc[:5]
    print(key.doc)
    for r in doc[:5]:
        print("\t{0}".format(r.arg1.gorn[:7]))
```

```
wsj_2315
    [0.0, 0.1.0, 0.1.1.0, 0.1.1.1, 0.1.1.2, 0.2]
    [1.1.1]
    [3]
    [5.1.1.1.0]
    [6.0, 6.1.0, 6.1.1.0, 6.1.1.1.0, 6.1.1.1.1, 6.1.1.1.2, 6.1.1.1.3.0]
wsj_2311
    [0]
wsj_2316
    [0.0.0, 0.0.1, 0.0.3, 0.1, 0.2]
    [2.0.0, 2.0.1, 2.0.3, 2.1, 2.2]
    [4]
    [5.3.4.1.1.2.2.2]
    [5.3.4]
```

2.3.6 Penn Treebank integration

```
from educe.pdtb import ptb

# confusingly, this is not an educe corpus reader, but the NLTK
# bracketed reader.  Sorry
ptb_reader = ptb.reader('{dd}/PTBIII/parsed/mrg/wsj/'.format(dd=data_dir))
ptb_trees = {}
for key in corpus.keys()[:3]:
    ptb_trees[key] = ptb.parse_trees(corpus, key, ptb_reader)
    print("{0}...".format(str(ptb_trees[key])[:100]))
```

```
[Tree('S', [Tree('NP-SBJ-1', [Tree('NNP', ['RJR']), Tree('NNP', ['Nabisco']), Tree(
↪'NNP', ['Inc.'])])]...
[Tree('S', [Tree('NP-SBJ', [Tree('NNP', ['CONCORDE']), Tree('JJ', ['trans-Atlantic']),
↪ Tree('NNS', [...
[Tree('S', [Tree('NP-SBJ', [Tree('NP', [Tree('DT', ['The']), Tree('NNP', ['U.S.'])]),␣
↪Tree(',', [','...
```

```
!ls ../data/PTBIII/parsed/mrg/wsj/
```

```
[34m00[m[m [34m01[m[m [34m02[m[m [34m03[m[m [34m04[m[m [34m05[m[m [34m06[m[m␣
↪[34m07[m[m [34m08[m[m [34m09[m[m [34m10[m[m [34m11[m[m [34m12[m[m [34m13[m[m␣
↪[34m14[m[m [34m15[m[m [34m16[m[m [34m17[m[m [34m18[m[m [34m19[m[m [34m20[m[m␣
↪[34m21[m[m [34m22[m[m [34m23[m[m [34m24[m[m
```

```
def pick_subtree(tree, gparts):
    if gparts:
        return pick_subtree(tree[gparts[0]], gparts[1:])
    else:
        return tree

# print the first seven gorn addresses for the first argument of the first
# 5 rels we read from each doc, along with the corresponding subtree
ndocs = 1
nrels = 3
ngorn = -1

for key in corpus.keys()[:1]:
```

```
    doc = corpus[key]
    rels = doc[:nrels]
    ptb_tree = ptb_trees[key]
    print("======="+key.doc)
    for i,r in enumerate(doc[:nrels]):
        print("---- relation {0}".format(i+1))
        print(display_rel(r))

        for (i,arg) in enumerate([r.arg1,r.arg2]):
            print(".... arg {0}".format(i+1))
            glist = arg.gorn # arg.gorn[:ngorn]
            subtrees = [pick_subtree(ptb_tree, g.parts) for g in glist]
            for gorn, subtree in zip(glist, subtrees):
                print("{0}\n{1}".format(gorn, str(subtree)))
```

```
=======wsj_2315
---- relation 1
[32mRJR Nabisco Inc. is disbanding its division responsible for buying␣
↪network advertising time[0m
     ---[[31mConnective(after | Temporal.Asynchronous.Succession)[0m]---->
                      [32mmoving 11 of the group's 14 employees to New York␣
↪from Atlanta[0m
.... arg 1
0.0
(NP-SBJ-1 (NNP RJR) (NNP Nabisco) (NNP Inc.))
0.1.0
(VBZ is)
0.1.1.0
(VBG disbanding)
0.1.1.1
(NP
  (NP (PRP$ its) (NN division))
  (ADJP
    (JJ responsible)
    (PP
      (IN for)
      (S-NOM
        (NP-SBJ (-NONE- ))
        (VP
          (VBG buying)
          (NP (NN network) (NN advertising) (NN time)))))))
0.1.1.2
(, ,)
0.2
(. .)
.... arg 2
0.1.1.3.2
(S-NOM
  (NP-SBJ (-NONE- *-1))
  (VP
    (VBG moving)
    (NP
      (NP (CD 11))
      (PP
```

```
        (IN of)
        (NP
          (NP (DT the) (NN group) (POS 's))
          (CD 14)
          (NNS employees))))
    (PP-DIR (TO to) (NP (NNP New) (NNP York)))
    (PP-DIR (IN from) (NP (NNP Atlanta)))))
---- relation 2
[32mthat it is shutting down the RJR Nabisco Broadcast unit, and dismissing␣
↪its 14 employees, in a move to save money[0m
     ---[Implicit [31mConnective(in addition | Expansion.Conjunction)[0m]---->
                    [32mRJR is discussing its network-buying plans with its␣
↪two main advertising firms, FCB/Leber Katz and McCann Erickson[0m
.... arg 1
1.1.1
(SBAR
  (IN that)
  (S
    (NP-SBJ (PRP it))
    (VP
      (VBZ is)
      (VP
        (VP
          (VBG shutting)
          (PRT (RP down))
          (NP
            (DT the)
            (NNP RJR)
            (NNP Nabisco)
            (NNP Broadcast)
            (NN unit)))
        (, ,)
        (CC and)
        (VP (VBG dismissing) (NP (PRP$ its) (CD 14) (NNS employees)))
        (, ,)
        (PP-LOC
          (IN in)
          (NP
            (DT a)
            (NN move)
            (S
              (NP-SBJ (-NONE- *))
              (VP (TO to) (VP (VB save) (NP (NN money))))))))))
.... arg 2
2.1.1
(SBAR
  (-NONE- 0)
  (S
    (NP-SBJ (NNP RJR))
    (VP
      (VBZ is)
      (VP
        (VBG discussing)
        (NP (PRP$ its) (JJ network-buying) (NNS plans))
```

```
        (PP
          (IN with)
          (NP
            (NP
              (PRP$ its)
              (CD two)
              (JJ main)
              (NN advertising)
              (NNS firms))
            (, ,)
            (NP
              (NP (NNP FCB/Leber) (NNP Katz))
              (CC and)
              (NP (NNP McCann) (NNP Erickson)))))))))
---- relation 3
[32mWe found with the size of our media purchases that an ad agency could do␣
↪just as good a job at significantly lower cost," said the spokesman, who␣
↪declined to specify how much RJR spends on network television time[0m
     ---[Entity]---->
                    [32mAn executive close to the company said RJR is␣
↪spending about $140 million on network television time this year, down from␣
↪roughly $200 million last year[0m
.... arg 1
3
(SINV
  (`` ``)
  (S-TPC-3
    (NP-SBJ (PRP We))
    (VP
      (VBD found)
      (PP
        (IN with)
        (NP
          (NP (DT the) (NN size))
          (PP (IN of) (NP (PRP$ our) (NNS media) (NNS purchases)))))
      (SBAR
        (IN that)
        (S
          (NP-SBJ (DT an) (NN ad) (NN agency))
          (VP
            (MD could)
            (VP
              (VB do)
              (NP (ADJP (RB just) (RB as) (JJ good)) (DT a) (NN job))
              (PP
                (IN at)
                (NP (ADJP (RB significantly) (JJR lower)) (NN cost)))))))))
  (, ,)
  ('' '')
  (VP (VBD said) (S (-NONE- *T-3)))
  (NP-SBJ
    (NP (DT the) (NN spokesman))
    (, ,)
    (SBAR
```

```
      (WHNP-1 (WP who))
      (S
        (NP-SBJ-4 (-NONE- *T*-1))
        (VP
          (VBD declined)
          (S
            (NP-SBJ (-NONE- *-4))
            (VP
              (TO to)
              (VP
                (VB specify)
                (SBAR
                  (WHNP-2 (WRB how) (JJ much))
                  (S
                    (NP-SBJ (NNP RJR))
                    (VP
                      (VBZ spends)
                      (NP (-NONE- *T*-2))
                      (PP-CLR
                        (IN on)
                        (NP (NN network) (NN television) (NN time))))))))))))
  (. .))
.... arg 2
4
(S
  (NP-SBJ
    (NP (DT An) (NN executive))
    (ADJP (RB close) (PP (TO to) (NP (DT the) (NN company)))))
  (VP
    (VBD said)
    (SBAR
      (-NONE- 0)
      (S
        (NP-SBJ (NNP RJR))
        (VP
          (VBZ is)
          (VP
            (VBG spending)
            (NP
              (NP
                (QP (RB about) ($ $) (CD 140) (CD million))
                (-NONE- *U*))
              (ADVP (-NONE- *ICH*-1)))
            (PP-CLR
              (IN on)
              (NP (NN network) (NN television) (NN time)))
            (NP-TMP (DT this) (NN year))
            (, ,)
            (ADVP-1
              (RB down)
              (PP
                (IN from)
                (NP
                  (NP
```

```
                    (QP (RB roughly) ($ $) (CD 200) (CD million))
                    (-NONE- *U*))
                  (NP-TMP (JJ last) (NN year))))))))))
  (. .))
```

```
print(subtree.flatten())
print(subtree.leaves())
```

```
(S
  An
  executive
  close
  to
  the
  company
  said
  0
  RJR
  is
  spending
  about
  $
  140
  million
  *U*
  *ICH*-1
  on
  network
  television
  time
  this
  year
  ,
  down
  from
  roughly
  $
  200
  million
  *U*
  last
  year
  .)
[u'An', u'executive', u'close', u'to', u'the', u'company', u'said', u'0',
 ↪ u'RJR', u'is', u'spending', u'about', u'$', u'140', u'million', u'*U*',␣
 ↪u'*ICH*-1', u'on', u'network', u'television', u'time', u'this', u'year', u',
 ↪', u'down', u'from', u'roughly', u'$', u'200', u'million', u'*U*', u'last',␣
 ↪u'year', u'.']
```

```
from copy import copy
t = copy(subtree)
print("constituent = "+ highlight(t.label()))
for i in range(len(subtree)):
    print(i)
    print(t.pop())
```

```
constituent = [31mS[0m
0
(. .)
1
(VP
  (VBD said)
  (SBAR
    (-NONE- 0)
    (S
      (NP-SBJ (NNP RJR))
      (VP
        (VBZ is)
        (VP
          (VBG spending)
          (NP
            (NP
              (QP (RB about) ($ $) (CD 140) (CD million))
              (-NONE- *U*))
            (ADVP (-NONE- *ICH*-1)))
          (PP-CLR
            (IN on)
            (NP (NN network) (NN television) (NN time)))
          (NP-TMP (DT this) (NN year))
          (, ,)
          (ADVP-1
            (RB down)
            (PP
              (IN from)
              (NP
                (NP
                  (QP (RB roughly) ($ $) (CD 200) (CD million))
                  (-NONE- *U*))
                (NP-TMP (JJ last) (NN year)))))))))
2
(NP-SBJ
  (NP (DT An) (NN executive))
  (ADJP (RB close) (PP (TO to) (NP (DT the) (NN company)))))
```

```
from copy import copy
t = copy(subtree)

def expand(subtree):
    if type(subtree) is unicode:
        print(subtree)
    else:
        print("constituent = "+ highlight(subtree.label()))
        for i, st in enumerate(subtree):
            #print(i)
            expand(st)

expand(t)
```

```
constituent = [31mS[0m
constituent = [31mNP-SBJ[0m
constituent = [31mNP[0m
```

```
constituent = [31mDT[0m
An
constituent = [31mNN[0m
executive
constituent = [31mADJP[0m
constituent = [31mRB[0m
close
constituent = [31mPP[0m
constituent = [31mTO[0m
to
constituent = [31mNP[0m
constituent = [31mDT[0m
the
constituent = [31mNN[0m
company
constituent = [31mVP[0m
constituent = [31mVBD[0m
said
constituent = [31mSBAR[0m
constituent = [31m-NONE-[0m
0
constituent = [31mS[0m
constituent = [31mNP-SBJ[0m
constituent = [31mNNP[0m
RJR
constituent = [31mVP[0m
constituent = [31mVBZ[0m
is
constituent = [31mVP[0m
constituent = [31mVBG[0m
spending
constituent = [31mNP[0m
constituent = [31mNP[0m
constituent = [31mQP[0m
constituent = [31mRB[0m
about
constituent = [31m$[0m
$
constituent = [31mCD[0m
140
constituent = [31mCD[0m
million
constituent = [31m-NONE-[0m
*U*
constituent = [31mADVP[0m
constituent = [31m-NONE-[0m
*ICH*-1
constituent = [31mPP-CLR[0m
constituent = [31mIN[0m
on
constituent = [31mNP[0m
constituent = [31mNN[0m
network
constituent = [31mNN[0m
```

```
television
constituent = [31mNN[0m
time
constituent = [31mNP-TMP[0m
constituent = [31mDT[0m
this
constituent = [31mNN[0m
year
constituent = [31m,[0m
,
constituent = [31mADVP-1[0m
constituent = [31mRB[0m
down
constituent = [31mPP[0m
constituent = [31mIN[0m
from
constituent = [31mNP[0m
constituent = [31mNP[0m
constituent = [31mQP[0m
constituent = [31mRB[0m
roughly
constituent = [31m$[0m
$
constituent = [31mCD[0m
200
constituent = [31mCD[0m
million
constituent = [31m-NONE-[0m
*U*
constituent = [31mNP-TMP[0m
constituent = [31mJJ[0m
last
constituent = [31mNN[0m
year
constituent = [31m.[0m
.
```

2.3.7 Work in progress

This tutorial is very much a work in progress. Moreover, support for the PDTB in educe is still very incomplete. So it's very much a moving target.

CHAPTER 3

Cookbook

Short how-tos on focused topics

3.1 [STAC] Turns and resources

Suppose you wanted to find the following (an actual request from the STAC project)

"Player offers to give resource X (possibly for Y) but does not hold resource X."

In this tutorial, we'll walk through such a query applying it to a single file in the corpus. Before digging into the tutorial proper, let's first read the sample data.

```
from __future__ import print_function
from educe.corpus import FileId
import educe.stac

# relative to the educe docs directory
data_dir = '../data'
corpus_dir = '{dd}/stac-sample'.format(dd=data_dir)

def text_snippet(text):
    "short text fragment"
    if len(text) < 43:
        return text
    else:
        return "{0}...{1}".format(text[:20], text[-20:])

def preview_unit(doc, anno):
    "the default str(anno) can be a bit overwhelming"
    preview = "{span: <11} {id: <20} [{type: <12}] {text}"
    text = doc.text(anno.text_span())
    return preview.format(id=anno.local_id(),
                          type=anno.type,
                          span=anno.text_span(),
```

```
                          text=text_snippet(text))

# pick out an example document to work with creating FileIds by hand
# is not something we would typically do (normally we would just iterate
# through a corpus), but it's useful for illustration
ex_key = FileId(doc='s1-league2-game3',
                subdoc='03',
                stage='units',
                annotator='BRONZE')
reader = educe.stac.Reader(corpus_dir)
ex_files = reader.filter(reader.files(),
                         lambda k: k == ex_key)
corpus = reader.slurp(ex_files, verbose=True)
ex_doc = corpus[ex_key]
```

```
Slurping corpus dir [1/1 done]
```

3.1.1 1. Turn and resource annotations

How would you go about doing it? One place to start is to look at turns and resources independently. We can filter turns and resources with the helper functions `is_turn` and `is_resource` from `educe.stac`

```
import educe.stac

ex_turns = [x for x in ex_doc.units if educe.stac.is_turn(x)]
ex_resources = [x for x in ex_doc.units if educe.stac.is_resource(x)]
ex_offers = [x for x in ex_resources if x.features['Status'] == 'Givable']

print("Example turns")
print("-------------")
for anno in ex_turns[:5]:
    # notice here that unit annotations have a features field
    print(preview_unit(ex_doc, anno))

print()
print("Example resources")
print("-----------------")
for anno in ex_offers[:5]:
    # notice here that unit annotations have a features field
    print(preview_unit(ex_doc, anno))
    print('', anno.features)
```

```
Example turns
-------------
(35,66)     stac_1368693098      [Turn        ] 152 : sabercat : yep, for what?
(100,123)   stac_1368693104      [Turn        ] 154 : sabercat : no way
(146,171)   stac_1368693110      [Turn        ] 156 : sabercat : could be
(172,191)   stac_1368693113      [Turn        ] 157 : amycharl : :)
(192,210)   stac_1368693116      [Turn        ] 160 : amycharl : ?

Example resources
-----------------
(84,88)     asoubeille_1374939917916 [Resource    ] clay
 {'Status': 'Givable', 'Kind': 'clay', 'Correctness': 'True', 'Quantity': '?'}
(141,144)   asoubeille_1374940096296 [Resource    ] ore
```

```
 {'Status': 'Givable', 'Kind': 'ore', 'Correctness': 'True', 'Quantity': '?'}
(398,403)   asoubeille_1374940373466 [Resource    ] sheep
 {'Status': 'Givable', 'Kind': 'sheep', 'Correctness': 'True', 'Quantity': '?'}
(464,467)   asoubeille_1374940434888 [Resource    ] ore
 {'Status': 'Givable', 'Kind': 'ore', 'Correctness': 'True', 'Quantity': '1'}
(689,692)   asoubeille_1374940671003 [Resource    ] one
 {'Status': 'Givable', 'Kind': 'Anaphoric', 'Correctness': 'True', 'Quantity': '1'}
```

Oh no, Anaphors

Oh dear, some of our resources won't tell us their types directly. They are anaphors pointing to other annotations. We'll ignore these for the moment, but it'll be important to deal with them properly later on.

3.1.2 2. Resources within turns?

It's not enough to be able to spit out resource and turn annotations.
What we really want to know about are which resources are within which

turns'

```
ex_turns_with_offers = [t for t in ex_turns if any(t.encloses(r) for r in ex_offers)]

print("Turns and resources within")
print("--------------------------")
for turn in ex_turns_with_offers[:5]:
    t_resources = [x for x in ex_resources if turn.encloses(x)]
    print(preview_unit(ex_doc, turn))
    for rsrc in t_resources:
        kind = rsrc.features['Kind']
        print("\t".join(["", str(rsrc.text_span()), kind]))
```

```
Turns and resources within
--------------------------
(959,1008)  stac_1368693191      [Turn        ] 201 : sabercat : can...or another␣
↪sheep? or
    (999,1004)      sheep
(1009,1030) stac_1368693195      [Turn        ] 202 : sabercat : two?
    (1026,1029)     Anaphoric
(67,99)     stac_1368693101      [Turn        ] 153 : amycharl : clay preferably
    (84,88) clay
(124,145)   stac_1368693107      [Turn        ] 155 : amycharl : ore?
    (141,144)       ore
(363,404)   stac_1368693135      [Turn        ] 171 : sabercat : want to trade for␣
↪sheep?
    (398,403)       sheep
```

3.1.3 3. But does the player own these resources?

Now that we can extract the resources within a turn, our next task is to figure out if the player actually has these resources to give. This information is stored in the turn features.

```
def parse_turn_resources(turn):
    """Return a dictionary of resource names to counts thereof
    """
    def split_eq(attval):
        key, val = attval.split('=')
        return key.strip(), int(val)
    rxs = turn.features['Resources']
    return dict(split_eq(x) for x in rxs.split(';'))

print("Turns and player resources")
print("--------------------------")
for turn in ex_turns[:5]:
    t_resources = [x for x in ex_resources if turn.encloses(x)]
    print(preview_unit(ex_doc, turn))
    # not to be confused with the resource annotations within the turn
    print('\t', parse_turn_resources(turn))
```

```
Turns and player resources
--------------------------
(35,66)     stac_1368693098      [Turn        ] 152 : sabercat : yep, for what?
     {'sheep': 5, 'wood': 2, 'ore': 2, 'wheat': 1, 'clay': 2}
(100,123)   stac_1368693104      [Turn        ] 154 : sabercat : no way
     {'sheep': 5, 'wood': 2, 'ore': 2, 'wheat': 1, 'clay': 2}
(146,171)   stac_1368693110      [Turn        ] 156 : sabercat : could be
     {'sheep': 5, 'wood': 2, 'ore': 2, 'wheat': 1, 'clay': 2}
(172,191)   stac_1368693113      [Turn        ] 157 : amycharl : :)
     {'sheep': 1, 'wood': 0, 'ore': 3, 'wheat': 1, 'clay': 3}
(192,210)   stac_1368693116      [Turn        ] 160 : amycharl : ?
     {'sheep': 1, 'wood': 1, 'ore': 2, 'wheat': 1, 'clay': 3}
```

3.1.4 4. Putting it together: is this an honest offer?

```
def is_somewhat_honest(turn, offer):
    """True if the player has the offered resource
    """
    if offer.features['Status'] != 'Givable':
        raise ValueError('Resource must be givable')
    kind = offer.features['Kind']
    t_rxs = parse_turn_resources(turn)
    return t_rxs.get(kind, 0) > 0

def is_honest(turn, offer):
    """
    True if the player has the offered resource
    at the quantity offered. Undefined for offers that
    do not have a defined quantity
    """
    if offer.features['Status'] != 'Givable':
        raise ValueError('Resource must be givable')
    if offer.features['Quantity'] == '?':
        raise ValueError('Resource must have a known quantity')
    promised = int(offer.features['Quantity'])
    kind = rsrc.features['Kind']
    t_rxs = parse_turn_resources(turn)
    return t_rxs.get(kind, 0) >= promised
```

```
def critique_offer(turn, offer):
    """Return some commentary on an offered resource"""
    kind = offer.features['Kind']
    quantity = offer.features['Quantity']
    honest = 'n/a' if quantity == '?' else is_honest(turn, offer)
    msg = ("\t{offered}/{has} {kind} | "
           "has some: {honestish}, "
           "enough: {honest}")
    return msg.format(kind=kind,
                      offered=quantity,
                      has=player_rxs.get(kind),
                      honestish=is_somewhat_honest(turn, offer),
                      honest=honest)

ex_turns_with_offers = [t for t in ex_turns if any(t.encloses(r) for r in ex_offers)]

print("Turns and offers")
print("----------------")
for turn in ex_turns_with_offers[:5]:
    offers = [x for x in ex_offers if turn.encloses(x)]
    print('', preview_unit(ex_doc, turn))
    player_rxs = parse_turn_resources(turn)
    for offer in offers:
        print(critique_offer(turn, offer))
```

```
Turns and offers
----------------
 (959,1008)  stac_1368693191      [Turn        ] 201 : sabercat : can...or another␣
↪sheep? or
    1/5 sheep | has some: True, enough: True
 (1009,1030) stac_1368693195      [Turn        ] 202 : sabercat : two?
    2/None Anaphoric | has some: False, enough: True
 (67,99)     stac_1368693101      [Turn        ] 153 : amycharl : clay preferably
    ?/3 clay | has some: True, enough: n/a
 (124,145)   stac_1368693107      [Turn        ] 155 : amycharl : ore?
    ?/3 ore | has some: True, enough: n/a
 (363,404)   stac_1368693135      [Turn        ] 171 : sabercat : want to trade for␣
↪sheep?
    ?/5 sheep | has some: True, enough: n/a
```

3.1.5 5. What about those anaphors?

Anaphors are represented with 'Anaphora' relation instances. Relation instances have a source and target connecting two unit level annotations (here two resources). The idea here is that the anaphor would be the source of the relation, and its antecedant is the target. We'll assume for simplicity that resource anaphora do not form chains.

```
import copy

resource_types = {}
for anno in ex_doc.relations:
    if anno.type != 'Anaphora':
        continue
    resource_types[anno.source] = anno.target.features['Kind']

print("Turns and offers (anaphors accounted for)")
print("-----------------------------------------")
```

```
for turn in ex_turns_with_offers[:5]:
    offers = [x for x in ex_offers if turn.encloses(x)]
    print('', preview_unit(ex_doc, turn))
    player_rxs = parse_turn_resources(turn)
    for offer in offers:
        if offer in resource_types:
            kind = resource_types[offer]
            offer = copy.copy(offer)
            offer.features['Kind'] = kind
        print(critique_offer(turn, offer))
```

```
Turns and offers (anaphors accounted for)
-----------------------------------------
 (959,1008)   stac_1368693191      [Turn        ] 201 : sabercat : can...or another␣
↪sheep? or
    1/5 sheep | has some: True, enough: True
 (1009,1030) stac_1368693195      [Turn        ] 202 : sabercat : two?
    2/5 sheep | has some: True, enough: True
 (67,99)     stac_1368693101      [Turn        ] 153 : amycharl : clay preferably
    ?/3 clay | has some: True, enough: n/a
 (124,145)   stac_1368693107      [Turn        ] 155 : amycharl : ore?
    ?/3 ore | has some: True, enough: n/a
 (363,404)   stac_1368693135      [Turn        ] 171 : sabercat : want to trade for␣
↪sheep?
    ?/5 sheep | has some: True, enough: n/a
```

3.1.6 Conclusion

In this tutorial, we've explored a couple of basic educe concepts, which we hope will enable you to extract some data from your discourse corpora, namely

- reading corpus data (and pre-filtering)
- standoff annotations
- searching by span enclosure, overlapping
- working with trees
- combining annotations from different sources

The concepts above should transfer to whatever discourse corpus you are working with (that educe supports, or that you are prepared to supply a reader for).

CHAPTER 4

educe package

Note: At the time of this writing, this is a slightly idealised representation of the package. See below for notes on where things get a bit messier

The educe library provides utilities for working with annotated discourse corpora. It has a three-layer structure:

- base layer (files, annotations, fusion, graphs)
- tool layer (specific to tools, file formats, etc)
- project layer (specific to particular corpora, currently stac)

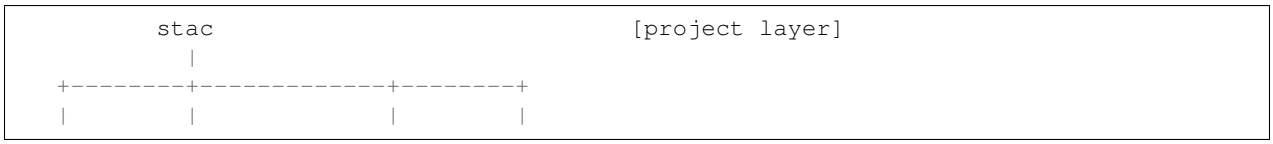
4.1 Layers

Working our way up the tower, the base layer provides four sublayers:

- file management (educe.corpus): basic model for corpus traversal, for selecting slices of the corpus
- annotation: (educe.annotation), representation of annotated texts, adhering closely to whatever annotation tool produced it.
- fusion (in progress): connections between annotations on different layers (eg. on speech acts for text spans, discourse relations), or from different tools (eg. from a POS tagger, a parser, etc)
- graph (educe.graph): high-level/abstract representation of discourse structure, allowing for queries on the structures themselves (eg. give me all pairs for discourse units separated by at most 3 nodes in the graph)

Building on the base layer, we have modules that are specific to a particular set of annotation tools, currently this is only *educe.glozz*. We aim to add modules sparingly.

Finally, on top of this, we have the project layer (eg. *educe.stac*) which keeps track of conventions specific to this particular corpus. The hope would be for most of your script writing to deal with this layer directly, eg. for STAC



```
   |        v             |        |
   |      glozz           |        |       [tool layer]
   |        |             |        |
   v        v             v        v
corpus -> annotation <- fusion <- graph    [base layer]
```

Support for other projects would consist in adding writing other project layer modules that map down to the tool layer.

4.2 Departures from the ideal (2013-05-23)

Educe is still its early stages. Some departures you may want to be aware of:

- fusion layer does not really exist yet; *educe.annotation* currently takes on some of the job (for example, the *text_span* function makes annotations of different types more or less comparable)
- layer violations: ideally we want lower layers to be abstract from things above them, but you may find eg. glozz-specific assumptions in the base layer, which isn't great.
- inconsistency in encapsulation: *educe.stac* doesn't wrap everything below it (it's also not clear yet if it should). It currently wraps educe.glozz and educe.corpus (so by rights you shouldn't really need to import them), but not the graph stuff for example.

4.3 Subpackages

4.3.1 educe.external package

Interacting with annotations from 3rd party tools

Submodules

educe.external.coref module

Coreference chain output in the form of educe standoff annotations (at least as emitted by Stanford's CoreNLP pipeline)

A coreference chain is considered to be a set of mentions. Each mention contains a set of tokens.

class `educe.external.coref.`**`Chain`**(*mentions*)
Bases: *educe.annotation.Standoff*

Chain of coreferences

class `educe.external.coref.`**`Mention`**(*tokens*, *head*, *most_representative=False*)
Bases: *educe.annotation.Standoff*

Mention of an entity

educe.external.corenlp module

Annotations from the CoreNLP pipeline

class educe.external.corenlp.**CoreNlpDocument**(*tokens*, *trees*, *deptrees*, *chains*)
Bases: *educe.annotation.Standoff*

All of the CoreNLP annotations for a particular document as instances of *educe.annotation.Standoff* or as structures that contain such instances.

class educe.external.corenlp.**CoreNlpToken**(*t*, *offset*, *origin=None*)
Bases: *educe.external.postag.Token*

A single token and its POS tag.

features
dict from str to str – Additional info found by corenlp about the token (eg. *x.features['lemma']*)

class educe.external.corenlp.**CoreNlpWrapper**(*corenlp_dir*)
Bases: object

Wrapper for the CoreNLP parsing system.

process(*txt_files*, *outdir*, *properties=[]*)
Run CoreNLP on text files

Parameters

- **txt_files** (*list of strings*) – Input files
- **outdir** (*string*) – Output dir
- **properties** (*list of strings, optional*) – Properties to control the behaviour of CoreNLP

Returns **corenlp_outdir** – Directory containing CoreNLP's output files

Return type string

educe.external.parser module

Syntactic parser output into educe standoff annotations (at least as emitted by Stanford's CoreNLP pipeline

This currently builds off the NLTK Tree class, but if the NLTK dependency proves too heavy, we could consider doing without.

class educe.external.parser.**ConstituencyTree**(*node*, *children*, *origin=None*)
Bases: *educe.external.parser.SearchableTree*, *educe.annotation.Standoff*

A variant of the NLTK Tree data structure which can be treated as an educe Standoff annotation.

This can be useful for representing syntactic parse trees in a way that can be later queried on the basis of Span enclosure.

Note that all children must have a *span* member of type *Span*

The *subtrees()* function can useful here.

classmethod build(*tree*, *tokens*)
Build an educe tree by combining an existing NLTK tree with some replacement leaves.

The replacement leaves should correspond 1:1 to the leaves of the original tree (for example, they may contain features related to those words).

Parameters

- **tree** (*nltk.Tree*) – Original NLTK tree.
- **tokens** (*iterable of Token*) – Sequence of replacement leaves.

Returns **ctree** – ConstituencyTree where the internal nodes have the same labels as in the original NLTK tree and the leaves correspond to the given sequence of tokens.

Return type *ConstituencyTree*

text_span()
Note: doc is ignored here

class educe.external.parser.**DependencyTree**(*node*, *children*, *link*, *origin=None*)
Bases: *educe.external.parser.SearchableTree*, *educe.annotation.Standoff*

A variant of the NLTK Tree data structure for the representation of dependency trees. The dependency tree is also considered a Standoff annotation but not quite in the same way that a constituency tree might be. The spans roughly indicate the range covered by the tokens in the subtree (this glosses over any gaps). They are mostly useful for determining if the tree (at its root node) pertains to any given sentence based on its offsets.

Fields:

- node is an some annotation of type *educe.annotation.Standoff*
- link is a string representing the link label between this node and its governor; None for the root node

classmethod build(*deps*, *nodes*, *k*, *link=None*)
Given two dictionaries

- mapping node ids to a list of (link label, child node id))
- mapping node ids to some representation of those nodes

and the id for the root node, build a tree representation of the dependency tree

is_root()
This is a dependency tree root (has a special node)

class educe.external.parser.**SearchableTree**(*node*, *children*)
Bases: nltk.tree.Tree

A tree with helper search functions

depth_first_iterator()
Iterate on the nodes of the tree, depth-first, pre-order.

topdown(*pred*, *prunable=None*)
Searching from the top down, return the biggest subtrees for which the predicate is True (or empty list if none are found).

The optional prunable function can be used to throw out subtrees for more efficient search (note that pred always overrides prunable though). Note that leaf nodes are ignored.

topdown_smallest(*pred*, *prunable=None*)
Searching from the top down, return the smallest subtrees for which the predicate is True (or empty list if none are found).

This is almost the same as *topdown*, except that if a subtree matches, we check for smaller matches in its subtrees.

Note that leaf nodes are ignored.

educe.external.postag module

CONLL formatted POS tagger output into educe standoff annotations (at least as emitted by CMU's ark-tweet-nlp.

Files are assumed to be UTF-8 encoded.

Note: NLTK has a CONLL reader too which looks a lot more general than this one

exception educe.external.postag.**EducePosTagException**(**args*, ***kw*)
Bases: exceptions.Exception

Exceptions that arise during POS tagging or when reading POS tag resources

class educe.external.postag.**RawToken**(*word*, *tag*)
Bases: object

A token with a part of speech tag associated with it

class educe.external.postag.**Token**(*tok*, *span*)
Bases: *educe.external.postag.RawToken*, *educe.annotation.Standoff*

A token with a part of speech tag and some character offsets associated with it.

classmethod left_padding()
Return a special Token for left padding

educe.external.postag.**generic_token_spans**(*text*, *tokens*, *offset=0*, *txtfn=None*)
Given a string and a sequence of substrings within than string, infer a span for each of the substrings.

We do this spans by walking the text and the tokens we consume substrings and skipping over any whitespace (including that which is within the tokens). For this to work, the substring sequence must be identical to the text modulo whitespace.

Spans are relative to the start of the string itself, but can be shifted by passing an offset (the start of the original string's span). Empty tokens are accepted but have a zero-length span.

Note: this function is lazy so you can use it incrementally provided you can generate the tokens lazily too

You probably want *token_spans* instead; this function is meant to be used for similar tasks outside of pos tagging

Parameters **txtfn** – function to extract text from a token (default None, treated as identity function)

educe.external.postag.**read_token_file**(*fname*)
Return a list of lists of RawToken

The input file format is what I believe to be the CONLL format (at least as emitted by the CMU Twitter POS tagger)

educe.external.postag.**token_spans**(*text*, *tokens*, *offset=0*)
Given a string and a sequence of RawToken representing tokens in that string, infer the span for each token. Return the results as a sequence of Token objects.

We infer these spans by walking the text as we consume tokens, and skipping over any whitespace in between. For this to work, the raw token text must be identical to the text modulo whitespace.

Spans are relative to the start of the string itself, but can be shifted by passing an offset (the start of the original string's span).

Parameters

- **text** (*str*) – Base text.
- **tokens** (*sequence of RawToken*) – Sequence of raw tokens in the text.
- **offset** (*int, defaults to 0*) – Offset for spans.

Returns **res** – Sequence of proper educe Tokens with their span.

Return type list of Token

educe.external.stanford_xml_reader module

Reader for Stanford CoreNLP pipeline outputs

Example of output:

```
<document>
  <sentences>
    <sentence id="1">
      <tokens>
      ...
      <token id="19">
      <word>direction</word>
      <lemma>direction</lemma>
      <CharacterOffsetBegin>135</CharacterOffsetBegin>
      <CharacterOffsetEnd>144</CharacterOffsetEnd>
      <POS>NN</POS>
      </token>
      <token id="20">
      <word>.</word>
      <lemma>.</lemma>
      <CharacterOffsetBegin>144</CharacterOffsetBegin>
      <CharacterOffsetEnd>145</CharacterOffsetEnd>
      <POS>.</POS>
      </token>
      ...
      <parse>(ROOT (S (PP (IN For) (NP (NP (DT a) (NN look)) (PP (IN at) (SBAR (WHNP␣
↪(WP what)) (S (VP (MD might) (VP (VB lie) (ADVP (RB ahead)) (PP (IN for) (NP (NNP U.
↪S.) (NNS forces))))))))))) (, ,) (VP (VB let) (S (NP (POS 's)) (VP (VB turn) (PP (TO␣
↪to) (NP (NP (PRP$ our) (NNP Miles) (NNP O'Brien)) (PP (IN in) (NP (NNP␣
↪Atlanta)))))))) (. .))) </parse>
      <basic-dependencies>
        <dep type="prep">
          <governor idx="13">let</governor>
          <dependent idx="1">For</dependent>
        </dep>
        ...
      </basic-dependencies>
      <collapsed-dependencies>
        <dep type="det">
          <governor idx="3">look</governor>
          <dependent idx="2">a</dependent>
        </dep>
        ...
      </collapsed-dependencies>
      <collapsed-ccprocessed-dependencies>
        <dep type="det">
          <governor idx="3">look</governor>
          <dependent idx="2">a</dependent>
        </dep>
        ...
      </collapsed-ccprocessed-dependencies>
    </sentence>
  </sentences>
</document>
```

IMPORTANT: Note that Stanford pipeline uses RHS inclusive offsets.

class educe.external.stanford_xml_reader.**PreprocessingSource**(*encoding='utf-8'*)

Bases: `object`

Reads in document annotations produced by CoreNLP pipeline.

This works as a stateful object that stores and provides access to all annotations contained in a CoreNLP output file, once the *read* method has been called.

get_coref_chains()
Get all coreference chains

get_document_id()
Get the document id

get_offset2sentence_map()
Get the offset to each sentence

get_offset2token_maps()
Get the offset to each token

get_ordered_sentence_list(*sort_attr='extent'*)
Get the list of sentences, ordered by sort_attr

get_ordered_token_list(*sort_attr='extent'*)
Get the list of tokens, ordered by sort_attr

get_sentence_annotations()
Get the annotations of all sentences

get_token_annotations()
Get the annotations of all tokens

read(*base_file*, *suffix='.raw.stanford'*)
Read and store the annotations from CoreNLP's output.

This function does not return anything, it modifies the state of the object to store the annotations.

educe.external.stanford_xml_reader.**test_file**(*base_filename*, *suffix='.raw.stanford'*)
Test that a file is effectively readable and print sentences

educe.external.stanford_xml_reader.**xml_unescape**(*_str*)
Get a proper string where special XML characters are unescaped.

Notes

You can also use *xml.sax.saxutils.escape*

4.3.2 educe.learning package

Submodules

educe.learning.csv module

educe.learning.edu_input_format module

This module implements a dumper for the EDU input format

See https://github.com/irit-melodi/attelo/blob/master/doc/input.rst

educe.learning.edu_input_format.**dump_all**(*X_gen*, *y_gen*, *f*, *class_mapping*, *docs*, *instance_generator*)

Dump a whole dataset: features (in svmlight) and EDU pairs.

Parameters

- **X_gen** (*iterable of int arrays*) – Feature vectors.
- **y_gen** (*iterable of int*) – Ground truth labels.
- **f** (*str*) – Output features file path
- **class_mapping** (*dict(str, int)*) – Mapping from label to int.
- **docs** (*list of DocumentPlus*) – Documents
- **instance_generator** (*function from doc to iterable of pairs*) – TODO

educe.learning.edu_input_format.**dump_edu_input_file**(*docs*, *f*)

Dump a dataset in the EDU input format.

Each document must have:

- edus: sequence of edu objects
- grouping: string (some sort of document id)
- edu2sent: int -> int or string or None (edu num to sentence num)

The EDUs must provide:

- identifier(): string
- text(): string

educe.learning.edu_input_format.**dump_pairings_file**(*epairs*, *f*)

Dump the EDU pairings

educe.learning.edu_input_format.**labels_comment**(*class_mapping*)

Return a string listing class labels in the format that attelo expects

educe.learning.edu_input_format.**load_labels**(*f*)

Read label set into a dictionary mapping labels to indices

educe.learning.keygroup_vectorizer module

This module provides ways to transform lists of PairKeys to sparse vectors.

class educe.learning.keygroup_vectorizer.**KeyGroupVectorizer**

Bases: object

Transforms lists of KeyGroups to sparse vectors.

vocabulary_

dict(str, int) – Vocabulary mapping.

fit_transform(*vectors*)

Learn the vocabulary dictionary and return instances

transform(*vectors*)

Transform documents to EDU pair feature matrix.

Extract features out of documents using the vocabulary fitted with fit.

educe.learning.keys module

Feature extraction keys.

A key is basically a feature name, its type, some help text.

We also provide a notion of groups that allow us to organise keys into sections

class `educe.learning.keys.`**Key**(*substance*, *name*, *description*)
Bases: `object`

Feature name plus a bit of metadata

classmethod basket(*name*, *description*)
A key for fields that represent a multiset of possible values. Baskets should be dictionaries from string to int (collections.Counter would be a good bet for collecting these)

classmethod continuous(*name*, *description*)
A key for fields that have range value (eg. numbers)

classmethod discrete(*name*, *description*)
A key for fields that have a finite set of possible values

substance = None
see *Substance*

class `educe.learning.keys.`**KeyGroup**(*description*, *keys*)
Bases: `dict`

A set of related features.

Note that a KeyGroup can be used as a dictionary, but instead of using Keys as values, you use the key names

DEBUG = True

NAME_WIDTH = 35

one_hot_values_gen(*suffix=''*)
Get a one-hot encoded version of this KeyGroups as a generator

suffix is added to the feature name

class `educe.learning.keys.`**MagicKey**(*substance*, *function*)
Bases: *educe.learning.keys.Key*

Somewhat fancier variant of Key that is built from a function The goal of the magic key is to reduce the amount of boilerplate needed to define keys

classmethod basket_fn(*function*)
A key for fields that represent a multiset of possible values. Baskets should be dictionaries from string to int (collections.Counter would be a good bet for collecting these)

classmethod continuous_fn(*function*)
A key for fields that have range value (eg. numbers)

classmethod discrete_fn(*function*)
A key for fields that have a finite set of possible values

class `educe.learning.keys.`**MergedKeyGroup**(*description*, *groups*)
Bases: *educe.learning.keys.KeyGroup*

A key group that is formed by fusing several key groups into one.

Note that for now all the keys in a merged group are lumped into the same object.

The help text tries to preserve the internal breakdown into the subgroups, however. It comes with a "level 1" section header, eg.

```
============================================================
big block of features
============================================================
```

class educe.learning.keys.**Substance**
Bases: object

The kind of the variable represented by this key.

- continuous
- discrete
- string (for meta vars; you probably want discrete instead)

If we ever reach a point where we're happy to switch to Python 3 wholesale, we should subclass Enum

BASKET = 4

CONTINUOUS = 1

DISCRETE = 2

STRING = 3

educe.learning.svmlight_format module

This module implements a dumper for the svmlight format

See *sklearn.datasets.svmlight_format*

educe.learning.svmlight_format.**dump_svmlight_file**(*X_gen*, *y_gen*, *f*, *zero_based=True*, *comment=None*, *query_id=None*)
Dump the dataset in svmlight file format.

educe.learning.util module

Common helper functions for feature extraction.

educe.learning.util.**space_join**(*str1*, *str2*)
join two strings with a space

educe.learning.util.**tuple_feature**(*combine*)

```
(a -> a -> b) ->
((current, cache, edu) -> a) ->
(current, cache, edu, edu) -> b)
```

Combine the result of single-edu feature function to make a pair feature

educe.learning.util.**underscore**(*str1*, *str2*)
join two strings with an underscore

educe.learning.vocabulary_format module

This module implements a loader and dumper for vocabularies.

`educe.learning.vocabulary_format.`**`dump_vocabulary`**(*vocabulary*, *f*)
Dump the vocabulary as a tab-separated file.

`educe.learning.vocabulary_format.`**`load_vocabulary`**(*f*)
Read vocabulary file into a dictionary of feature name and index

4.3.3 educe.pdtb package

Conventions specific to the Penn Discourse Treebank (PDTB) project

Subpackages

educe.pdtb.util package

Submodules

educe.pdtb.util.args module

Command line options

`educe.pdtb.util.args.`**`add_usual_input_args`**(*parser*)
Augment a subcommand argparser with typical input arguments. Sometimes your subcommand may require slightly different output arguments, in which case, just don't call this function.

`educe.pdtb.util.args.`**`add_usual_output_args`**(*parser*)
Augment a subcommand argparser with typical output arguments, Sometimes your subcommand may require slightly different output arguments, in which case, just don't call this function.

`educe.pdtb.util.args.`**`announce_output_dir`**(*output_dir*)
Tell the user where we saved the output

`educe.pdtb.util.args.`**`get_output_dir`**(*args*)
Return the output directory specified on (or inferred from) the command line arguments, *creating it if necessary*.

We try the following in order:

1. If *–output* is given explicitly, we'll just use/create that
2. OK just make a temporary directory. Later on, you'll probably want to call *announce_output_dir*.

`educe.pdtb.util.args.`**`mk_output_path`**(*odir*, *k*)
Path stub (needs extension) given an output directory and a PDTB corpus key

`educe.pdtb.util.args.`**`read_corpus`**(*args*, *verbose=True*)
Read the section of the corpus specified in the command line arguments.

educe.pdtb.util.features module

Feature extraction library functions for PDTB corpus

class `educe.pdtb.util.features.`**`DocumentPlus`**(*key*, *doc*)
Bases: `tuple`

doc
Alias for field number 1

key
Alias for field number 0

class educe.pdtb.util.features.**FeatureInput**(*corpus*, *debug*)
Bases: tuple

corpus
Alias for field number 0

debug
Alias for field number 1

class educe.pdtb.util.features.**RelKeys**(*inputs*)
Bases: *educe.learning.keys.MergedKeyGroup*

Features for relations

fill(*current*, *rel*, *target=None*)
See *RelSubgroup*

class educe.pdtb.util.features.**RelSubGroup_Core**
Bases: *educe.pdtb.util.features.RelSubgroup*

core features

fill(*current*, *rel*, *target=None*)

class educe.pdtb.util.features.**RelSubgroup**(*description*, *keys*)
Bases: *educe.learning.keys.KeyGroup*

Abstract keygroup for subgroups of the merged RelKeys. We use these subgroup classes to help provide modularity, to capture the idea that the bits of code that define a set of related feature vector keys should go with the bits of code that also fill them out

fill(*current*, *rel*, *target=None*)
Fill out a vector's features (if the vector is None, then we just fill out this group; but in the case of a merged key group, you may find it desirable to fill out the merged group instead)

class educe.pdtb.util.features.**SingleArgKeys**(*inputs*)
Bases: *educe.learning.keys.MergedKeyGroup*

Features for a single EDU

fill(*current*, *arg*, *target=None*)
See *SingleArgSubgroup.fill*

class educe.pdtb.util.features.**SingleArgSubgroup**(*description*, *keys*)
Bases: *educe.learning.keys.KeyGroup*

Abstract keygroup for subgroups of the merged SingleArgKeys. We use these subgroup classes to help provide modularity, to capture the idea that the bits of code that define a set of related feature vector keys should go with the bits of code that also fill them out

fill(*current*, *arg*, *target=None*)
Fill out a vector's features (if the vector is None, then we just fill out this group; but in the case of a merged key group, you may find it desirable to fill out the merged group instead)

educe.pdtb.util.features.**extract_rel_features**(*inputs*)
Return a pair of dictionaries, one for attachments and one for relations

educe.pdtb.util.features.**mk_current**(*inputs*, *k*)
Pre-process and bundle up a representation of the current document

educe.pdtb.util.features.**spans_to_str**(*spans*)
string representation of a list of spans, meant to work as an id

Submodules

educe.pdtb.corpus module

PDTB Corpus management (re-exported by educe.pdtb)

class educe.pdtb.corpus.**Reader**(*corpusdir*)
Bases: *educe.corpus.Reader*

See *educe.corpus.Reader* for details

files(*doc_glob=None*)

Parameters **doc_glob** (*str, optional*) – Glob expression for document (folder) names ; if *None*, it uses the wildcard '/' for folder names and file basenames.

slurp_subcorpus(*cfiles*, *verbose=False*)
See *educe.rst_dt.parse* for a description of *RSTTree*

educe.pdtb.corpus.**id_to_path**(*k*)
Given a fleshed out FileId (none of the fields are None), return a filepath for it following Penn Discourse Treebank conventions.

You will likely want to add your own filename extensions to this path

educe.pdtb.corpus.**mk_key**(*doc*)
Return an corpus key for a given document name

educe.pdtb.parse module

Standalone parser for PDTB files.

The function *parse* takes a single .pdtb file and returns a list of *Relation*, with the following subtypes:

Relation	selection	features	sup?
ExplicitRelation	*Selection*	attr, 1 connhead	Y
ImplicitRelation	*InferenceSite*	attr, 2 conn	Y
AltLexRelation	*Selection*	attr, 2 semclass	Y
EntityRelation	*InferenceSite*	none	N
NoRelation	*InferenceSite*	none	N

These relation subtypes are stitched together (and inherit members) from two or three components

- arguments: always *arg1* and *arg2*; but in some cases, the arguments can have supplementary information
- selection: see either *Selection* or *InferenceSite*
- some features (see eg. *ExplictRelationFeatures*)

The simplest way to get to grips with this may be to try the *parse* function on some sample relations and print the resulting objects.

class educe.pdtb.parse.**AltLexRelation**(*selection*, *features*, *args*)
Bases: *educe.pdtb.parse.Selection*, *educe.pdtb.parse.AltLexRelationFeatures*, *educe.pdtb.parse.Relation*

class educe.pdtb.parse.**AltLexRelationFeatures**(*attribution*, *semclass1*, *semclass2*)
Bases: *educe.pdtb.parse.PdtbItem*

class educe.pdtb.parse.**Arg**(*selection*, *attribution=None*, *sup=None*)
Bases: *educe.pdtb.parse.Selection*

class educe.pdtb.parse.**Attribution**(*source*, *type*, *polarity*, *determinacy*, *selection=None*)
Bases: *educe.pdtb.parse.PdtbItem*

class educe.pdtb.parse.**Connective**(*text*, *semclass1*, *semclass2=None*)
Bases: *educe.pdtb.parse.PdtbItem*

class educe.pdtb.parse.**EntityRelation**(*infsite*, *args*)
Bases: *educe.pdtb.parse.InferenceSite*, *educe.pdtb.parse.Relation*

class educe.pdtb.parse.**ExplicitRelation**(*selection*, *features*, *args*)
Bases: *educe.pdtb.parse.Selection*, *educe.pdtb.parse.ExplicitRelationFeatures*, *educe.pdtb.parse.Relation*

class educe.pdtb.parse.**ExplicitRelationFeatures**(*attribution*, *connhead*)
Bases: *educe.pdtb.parse.PdtbItem*

class educe.pdtb.parse.**GornAddress**(*parts*)
Bases: *educe.pdtb.parse.PdtbItem*

class educe.pdtb.parse.**ImplicitRelation**(*infsite*, *features*, *args*)
Bases: *educe.pdtb.parse.InferenceSite*, *educe.pdtb.parse.ImplicitRelationFeatures*, *educe.pdtb.parse.Relation*

class educe.pdtb.parse.**ImplicitRelationFeatures**(*attribution*, *connective1*, *connective2=None*)
Bases: *educe.pdtb.parse.PdtbItem*

class educe.pdtb.parse.**InferenceSite**(*strpos*, *sentnum*)
Bases: *educe.pdtb.parse.PdtbItem*

class educe.pdtb.parse.**NoRelation**(*infsite*, *args*)
Bases: *educe.pdtb.parse.InferenceSite*, *educe.pdtb.parse.Relation*

class educe.pdtb.parse.**PdtbItem**
Bases: object

class educe.pdtb.parse.**Relation**(*args*)
Bases: *educe.pdtb.parse.PdtbItem*

arg1
TODO – TODO

arg2
TODO – TODO

class educe.pdtb.parse.**Selection**(*span*, *gorn*, *text*)
Bases: *educe.pdtb.parse.PdtbItem*

class educe.pdtb.parse.**SemClass**(*klass*)
Bases: *educe.pdtb.parse.PdtbItem*

class educe.pdtb.parse.**Sup**(*selection*)
Bases: *educe.pdtb.parse.Selection*

educe.pdtb.parse.**parse**(*path*)
Retrieve the list of relations found in a single .pdtb file.

Parameters **path** (*str*) – Path to the .pdtb file (?)

Returns **relations** – List of relations found.

Return type list of Relation

`educe.pdtb.parse.`**`parse_relation`**(*s*)
Parse a single relation or throw a ParseException.

`educe.pdtb.parse.`**`split_relations`**(*s*)

educe.pdtb.pdtbx module

PDTB in an adhoc (educe-grown) XML format, unfortunately not a standard, but a little homegrown language using XML syntax. I'll call it pdtbx. No reason it can't be used outside of educe.

Informal DTD:

- SpanList is attribute spanList in PDTB string convention
- GornAddressList is attribute gornList in PDTB string convention
- **SemClass is attribute semclass1 (and optional attribute semclass2)** in PDTB string convention
- text in <*text*> elements with usual XML escaping conventions
- args in <*arg*> elements in order (arg1 before arg2)
- implicitRelations can have multiple connectives

`educe.pdtb.pdtbx.`**`Relation_xml`**(*itm*)

`educe.pdtb.pdtbx.`**`Relations_xml`**(*itms*)

`educe.pdtb.pdtbx.`**`read_Relation`**(*node*)

`educe.pdtb.pdtbx.`**`read_Relations`**(*node*)

`educe.pdtb.pdtbx.`**`read_pdtbx_file`**(*filename*)

`educe.pdtb.pdtbx.`**`write_pdtbx_file`**(*filename*, *relations*)

educe.pdtb.ptb module

Alignment with the Penn Treebank

`educe.pdtb.ptb.`**`parse_trees`**(*corpus*, *k*, *ptb*)
Given an PDTB document and an NLTK PTB reader, return the PTB trees.

Note that a future version of this function will try to educify the trees as well, but for now things will be fairly rudimentary

`educe.pdtb.ptb.`**`reader`**(*corpus_dir*)
An instantiated NLTK BracketedParseCorpusReader for the PTB section relevant to the PDTB corpus.

Note that the path you give to this will probably end with something like *parsed/mrg/wsj*

4.3.4 educe.ptb package

Conventions specific to the Penn Treebank.

The PTB isn't a discourse corpus as such, but a supplementary resource to be combined with the RST DT or the PDTB

Submodules

educe.ptb.annotation module

Educe representation of Penn Tree Bank annotations.

We actually just use the token and constituency tree representations from *educe.external.postag* and *educe.external.parse*, but included here are tools that can also be used to align the PTB with other corpora based off the same text (eg. the RST Discourse Treebank)

educe.ptb.annotation.**PTB_TO_TEXT = {“’”: ””, “‘’: ””, ‘-LSB-‘: ‘[’, ‘-RRB-‘: ‘)’, ‘-LCB-‘: ‘{‘, ‘-LRB-‘: ‘(‘, ‘-RSB-‘:**
Straight substitutions you can use to replace some PTB-isms with their likely original text

class educe.ptb.annotation.**TweakedToken**(*word*, *tag*, *tweaked_word=None*, *prefix=None*)
Bases: *educe.external.postag.RawToken*

A token with word, part of speech, plus “tweaked word” (what the token should be treated as when aligning with corpus), and offset (some tokens should skip parts of the text)

This intermediary class should only be used within the educe library itself. The context is that we sometimes want to align PTB annotations (see *educe.external.postag.generic_token_spans*) against text which is almost but not quite identical to the text that PTB annotations seem to represent. For example, the source text might have sentences that end in abbreviations, like “He moved to the U.S.” and the PTB might annotation an extra full stop after this for an end-of-sentence marker. To deal with these, we use wrapped tokens to allow for some manual substitutions:

- you could “delete” a token by assigning it an empty tweaked word (it would then be assigned a zero-length span)
- you could skip some part of the text by supplying a prefix (this expands the tweaked word, and introduces an offset which you can subsequentnly use to adjust the detected token span)
- or you could just replace the token text outright

These tweaked tokens are only used to obtain a span within the text you are trying to align against; they can be subsequently discarded.

educe.ptb.annotation.**basic_category**(*label*)
Get the basic syntactic category of a label.

This is done by truncating whatever comes after a (non-word-initial) occurrence of one of the label_annotation_introducing_characters().

educe.ptb.annotation.**is_empty_category**(*postag*)
True if postag is the empty category, i.e. *-NONE-* in the PTB.

educe.ptb.annotation.**is_non_empty**(*tree*)
Filter (return False for) nodes that cover a totally empty span.

educe.ptb.annotation.**is_nonword_token**(*text*)
True if the text appears to correspond to some kind of non-textual token, for example, **T*-1* for some kind of trace. These seem to only appear with tokens tagged *-NONE-*.

educe.ptb.annotation.**post_basic_category_index**(*label*)
Get the index of the first char after the basic label.

This should never match the first char of the label ; if the first char is such a char, then a matched char is also not used iff there is something in between, e.g. (-LRB- => -LRB-) but (–PU => -).

educe.ptb.annotation.**prune_tree**(*tree*, *filter_func*)
Prune a tree by applying filter_func recursively.

All children of filtered nodes are pruned as well. Nodes whose children have all been pruned are pruned too.

The filter function must be applicable to Tree but also non-Tree, as are leaves in an NLTK Tree.

educe.ptb.annotation.**strip_punctuation**(*tokens*)
Strip leading and trailing punctuation from a sequence of tokens.

Parameters **tokens** (*list of Token*) – Sequence of tokens.

Returns **tokens_strip** – Corresponding list of tokens with no leading or trailing punctuation.

Return type list of Token

educe.ptb.annotation.**strip_subcategory**(*tree*, *retain_TMP_subcategories=False*, *retain_NPTMP_subcategories=False*)
Transform tree to strip additional label annotation at each node

educe.ptb.annotation.**syntactic_node_seq**(*ptree*, *tokens*)
Find the sequence of syntactic nodes covering a sequence of tokens.

Parameters

- **ptree** (*nltk.tree.Tree*) – Syntactic tree.
- **tokens** (sequence of *Token*) – Sequence of tokens under scrutiny.

Returns **syn_nodes** – Spanning sequence of nodes of the syntactic tree.

Return type list of *nltk.tree.Tree*

educe.ptb.annotation.**transform_tree**(*tree*, *transformer*)
Transform a tree by applying a transformer at each level.

The tree is traversed depth-first, left-to-right, and the transformer is applied at each node.

educe.ptb.head_finder module

This submodule provides several functions that find heads in trees.

It uses head rules as described in (Collins 1999), Appendix A. See *http://www.cs.columbia.edu/~mcollins/papers/heads*, Bikel's 2004 CL paper on the intricacies of Collins' parser and the classes in (StanfordNLP) CoreNLP that inherit from *AbstractCollinsHeadFinder.java* .

educe.ptb.head_finder.**find_edu_head**(*tree*, *hwords*, *wanted*)
Find the head word of a set of wanted nodes from a tree.

The tree is traversed top-down, breadth first, until we reach a node headed by a word from *wanted*.

Return a pair of treepositions (head node, head word), or None if no occurrence of any word in *wanted* was found.

This function is typically called for each EDU, *wanted* being the set of tree positions of its tokens, after *find_lexical_heads* has been called on the entire *tree* (providing *hwords*).

Parameters

- **tree** (*nltk.Tree* with *educe.external.postag.RawToken* leaves) – PTB tree whose lexical heads we want.
- **hwords** (*dict(tuple(int), tuple(int))*) – Map from each node of the constituency tree to its lexical head. Both nodes are designated by their (NLTK) tree position (a.k.a. Gorn address).
- **wanted** (*iterable of tuple(int)*) – The tree positions of the tokens in the span of interest, e.g. in the EDU we are looking at.

Returns

- **cur_treepos** (*tuple(int)*) – Tree position of the head node, i.e. the highest node headed by a word from wanted.
- **cur_hw** (*tuple(int)*) – Tree position of the head word.

`educe.ptb.head_finder.`**`find_lexical_heads`**(*tree*)
Find the lexical head at each node of a constituency tree.

The logic corresponds to Collins' head finding rules.

This is typically used to find the lexical head of each node of a (clean) *educe.external.parser.ConstituencyTree* whose leaves are *educe.external.postag.Token*.

Parameters **`tree`** (*nltk.Tree* with *educe.external.postag.RawToken* leaves) – PTB tree whose lexical heads we want

Returns **head_word** – Map each node of the constituency tree to its lexical head. Both nodes are designated by their (NLTK) tree position (a.k.a. Gorn address).

Return type dict(tuple(int), tuple(int))

`educe.ptb.head_finder.`**`load_head_rules`**(*f*)
Load the head rules from file f.

Return a dictionary from parent non-terminal to (direction, priority list).

4.3.5 educe.rst_dt package

Conventions specific to the RST discourse treebank project

Subpackages

educe.rst_dt.learning package

Submodules

educe.rst_dt.learning.args module

Command line options for learning commands

class `educe.rst_dt.learning.args.`**`FeatureSetAction`**(*option_strings*, *dest*, *nargs=None*, ***kwargs*)

Bases: `argparse.Action`

Select the desired feature set

`educe.rst_dt.learning.args.`**`add_usual_input_args`**(*parser*)
Augment a subcommand argparser with typical input arguments. Sometimes your subcommand may require slightly different output arguments, in which case, just don't call this function.

educe.rst_dt.learning.base module

Basics for feature extraction

class `educe.rst_dt.learning.base.`**`DocumentPlusPreprocessor`**(*token_filter=None*, *word2clust=None*)

Bases: `object`

Preprocessor for feature extraction on a DocumentPlus

This pre-processor currently does not explicitly impute missing values, but it probably should eventually. As the ultimate output is features in a sparse format, the current strategy amounts to imputing missing values as 0, which is most certainly not optimal.

preprocess(*doc*, *strict=False*)

Preprocess a document and output basic features for each EDU.

Parameters **doc** (DocumentPlus) – Document to be processed.

Returns

- **edu_infos** (*list of dict of features*) – List of basic features for each EDU ; each feature is a couple (basic_feat_name, basic_feat_val).
- **para_infos** (*list of dict of features*) – List of basic features for each paragraph ; each feature is a couple (basic_feat_name, basic_feat_val).

exception educe.rst_dt.learning.base.**FeatureExtractionException**(*msg*)

Bases: exceptions.Exception

Exceptions related to RST trees not looking like we would expect them to

educe.rst_dt.learning.base.**edu_feature**(*wrapped*)

Lift a function from *edu -> feature* to *single_function_input -> feature*

educe.rst_dt.learning.base.**edu_pair_feature**(*wrapped*)

Lifts a function from *(edu, edu) -> f* to *pair_function_input -> f*

educe.rst_dt.learning.base.**lowest_common_parent**(*treepositions*)

Find tree position of the lowest common parent of a list of nodes.

Parameters **treepositions** (list of tree positions) – see nltk.tree.Tree.treepositions()

Returns **tpos_parent** – Tree position of the lowest common parent to all the given tree positions.

Return type tree position

educe.rst_dt.learning.base.**on_first_bigram**(*wrapped*)

Lift a function from *a -> string* to *[a] -> string* the function will be applied to the up to first two elements of the list and the result concatenated. It returns None if the list is empty

educe.rst_dt.learning.base.**on_first_unigram**(*wrapped*)

Lift a function from *a -> b* to *[a] -> b* taking the first item or returning None if empty list

educe.rst_dt.learning.base.**on_last_bigram**(*wrapped*)

Lift a function from *a -> string* to *[a] -> string* the function will be applied to the up to the two elements of the list and the result concatenated. It returns None if the list is empty

educe.rst_dt.learning.base.**on_last_unigram**(*wrapped*)

Lift a function from *a -> b* to *[a] -> b* taking the last item or returning None if empty list

educe.rst_dt.learning.doc_vectorizer module

This submodule implements document vectorizers

class educe.rst_dt.learning.doc_vectorizer.**DocumentCountVectorizer**(*instance_generator*, *feature_set*, *lexsie_data_dir=None*, *max_df=1.0*, *min_df=1*, *max_features=None*, *vocabulary=None*, *separator='='*, *split_feat_space=None*)

Bases: object

Fancy vectorizer for the RST-DT treebank.

See *sklearn.feature_extraction.text.CountVectorizer* for reference.

build_analyzer()
Return a callable that extracts feature vectors from a doc

decode(*doc*)
Decode the input into a DocumentPlus.

Currently a no-op except for type checking.

Parameters **doc** (educe.rst_dt.document_plus.DocumentPlus) – Rich representation of the document.

Returns **doc** – Rich representation of the document.

Return type *educe.rst_dt.document_plus.DocumentPlus*

fit(*raw_documents*, *y=None*)
Learn a vocabulary dictionary of all features from the documents

fit_transform(*raw_documents*, *y=None*)
Learn the vocabulary dictionary and generate a feature matrix per document.

transform(*raw_documents*)
Transform documents to a feature matrix.

Generate a feature matrix, one row per instance.

Parameters **raw_documents** (*TODO*) – TODO

Yields **row** (*(row, (tgt, src))*) – Feature vector for the next instance.

class educe.rst_dt.learning.doc_vectorizer.**DocumentLabelExtractor**(*instance_generator*, *ordered_pairs=True*, *unknown_label='__UNK__'*, *labelset=None*)

Bases: object

Label extractor for the RST-DT treebank.

fixed_labelset_
boolean – True if the labelset has been fixed, i.e. *self* has been fit.

labelset_
dict – A mapping of labels to indices.

build_analyzer()
Return a callable that extracts feature vectors from a doc

decode(*doc*)
Currently a no-op if doc is a DocumentPlus.

Raises an exception otherwise. Was: Decode the input into a DocumentPlus.

Parameters **doc** (DocumentPlus) – Rich representation of the document.

Returns **doc** – Rich representation of *doc*.

Return type *DocumentPlus*

fit(*raw_documents*)
Learn a labelset from the documents

fit_transform(*raw_documents*)
Learn the label encoder and return a vector of labels

There is one label per instance extracted from raw_documents.

transform(*raw_documents*)
Transform documents to a label vector

educe.rst_dt.learning.doc_vectorizer.**re_emit**(*feats*, *suff*)
Re-emit feats with suff appended to each feature name

educe.rst_dt.learning.features module

Feature extraction library functions for RST_DT corpus

educe.rst_dt.learning.features.**build_doc_preprocessor**()
Build the preprocessor for feature extraction in each EDU of doc

educe.rst_dt.learning.features.**build_edu_feature_extractor**()
Build the feature extractor for single EDUs

educe.rst_dt.learning.features.**build_pair_feature_extractor**()
Build the feature extractor for pairs of EDUs

TODO: properly emit features on single EDUs ; they are already stored in sf_cache, but under (slightly) different names

educe.rst_dt.learning.features.**combine_features**(*feats_g*, *feats_d*, *feats_gd*)
Generate features by taking a (linear) combination of features.

I suspect these do not have a great impact, if any, on results.

Parameters

- **feats_g** (*dict(feat_name, feat_val)*) – features of the gov EDU
- **feats_d** (*dict(feat_name, feat_val)*) – features of the dep EDU
- **feats_gd** (*dict(feat_name, feat_val)*) – features of the (gov, dep) edge

Returns **cf** – combined features

Return type dict(feat_name, feat_val)

educe.rst_dt.learning.features.**extract_pair_gap**(*doc*, *edu_info1*, *edu_info2*)
Document tuple features

educe.rst_dt.learning.features.**extract_pair_pos_tags**(*doc*, *edu_info1*, *edu_info2*)
POS tag features on EDU pairs

educe.rst_dt.learning.features.**extract_pair_raw_word**(*doc*, *edu_info1*, *edu_info2*)
raw word features on EDU pairs

educe.rst_dt.learning.features.**extract_single_ptb_token_pos**(*doc*, *edu_info*, *para_info*)
POS features on PTB tokens for the EDU

educe.rst_dt.learning.features.**extract_single_ptb_token_word**(*doc*, *edu_info*, *para_info*)
word features on PTB tokens for the EDU

educe.rst_dt.learning.features.**extract_single_raw_word**(*doc*, *edu_info*, *para_info*)
raw word features for the EDU

educe.rst_dt.learning.features.**product_features**(*feats_g*, *feats_d*, *feats_gd*)
Generate features by taking the product of features.

Parameters

- **feats_g** (*dict(feat_name, feat_val)*) – features of the gov EDU
- **feats_d** (*dict(feat_name, feat_val)*) – features of the dep EDU
- **feats_gd** (*dict(feat_name, feat_val)*) – features of the (gov, dep) edge

Returns **pf** – product features

Return type dict(feat_name, feat_val)

educe.rst_dt.learning.features_dev module

Experimental features.

class educe.rst_dt.learning.features_dev.**LecsieFeats**(*lecsie_data_dir*)
Bases: object

Extract Lecsie features from each pair of EDUs

fit(*edu_pairs*, *y=None*)
Fit the feature extractor.

Currently a no-op.

Parameters

- **edu_pairs** (*TODO*) – TODO
- **y** (*TODO, optional*) – TODO

Returns **self** – TODO

Return type TODO

transform(*edu_pairs*)
Extract lecsie features for pairs of EDUs.

This is a generator.

Parameters **edu_pairs** (*TODO*) – TODO

Returns **res** – TODO

Return type TODO

educe.rst_dt.learning.features_dev.**build_doc_preprocessor**()
Build the preprocessor for feature extraction in each EDU of doc

educe.rst_dt.learning.features_dev.**build_edu_feature_extractor**()
Build the feature extractor for single EDUs

educe.rst_dt.learning.features_dev.**build_pair_feature_extractor**(*lecsie_data_dir=None*)
Build the feature extractor for pairs of EDUs

TODO: properly emit features on single EDUs ; they are already stored in sf_cache, but under (slightly) different names

educe.rst_dt.learning.features_dev.**combine_features**(*feats_g*, *feats_d*, *feats_gd*)
Generate features by taking a (linear) combination of features.

I suspect these do not have a great impact, if any, on results.

Parameters

- **feats_g** (*dict(feat_name, feat_val)*) – features of the gov EDU
- **feats_d** (*dict(feat_name, feat_val)*) – features of the dep EDU
- **feats_gd** (*dict(feat_name, feat_val)*) – features of the (gov, dep) edge

Returns **cf** – combined features

Return type dict(feat_name, feat_val)

educe.rst_dt.learning.features_dev.**extract_pair_doc**(*doc*, *edu_info1*, *edu_info2*, *edu_info_bwn*)
Document-level tuple features

educe.rst_dt.learning.features_dev.**extract_pair_para**(*doc*, *edu_info1*, *edu_info2*, *edu_info_bwn*)
Paragraph tuple features

educe.rst_dt.learning.features_dev.**extract_pair_sent**(*doc*, *edu_info1*, *edu_info2*, *edu_info_bwn*)
Sentence tuple features

educe.rst_dt.learning.features_dev.**extract_pair_syntax**(*doc*, *edu_info1*, *edu_info2*, *edu_info_bwn*)
syntactic features for the pair of EDUs

educe.rst_dt.learning.features_dev.**extract_single_brown**(*doc*, *edu_info*, *para_info*)
Brown cluster features for the EDU

educe.rst_dt.learning.features_dev.**extract_single_length**(*doc*, *edu_info*, *para_info*)
Sentence features for the EDU

educe.rst_dt.learning.features_dev.**extract_single_para**(*doc*, *edu_info*, *para_info*)
paragraph features for the EDU

educe.rst_dt.learning.features_dev.**extract_single_pdtb_markers**(*doc*, *edu_info*, *para_info*)
Features on the presence of PDTB discourse markers in the EDU

educe.rst_dt.learning.features_dev.**extract_single_pos**(*doc*, *edu_info*, *para_info*)
POS features for the EDU

educe.rst_dt.learning.features_dev.**extract_single_sentence**(*doc*, *edu_info*, *para_info*)
Sentence features for the EDU

educe.rst_dt.learning.features_dev.**extract_single_syntax**(*doc*, *edu_info*, *para_info*)
syntactic features for the EDU

educe.rst_dt.learning.features_dev.**extract_single_typo**(*doc*, *edu_info*, *para_info*)
typographical features for the EDU

educe.rst_dt.learning.features_dev.**extract_single_word**(*doc*, *edu_info*, *para_info*)
word features for the EDU

educe.rst_dt.learning.features_dev.**is_title_cased**(*tok_seq*)
True if a sequence of tokens is title-cased

educe.rst_dt.learning.features_dev.**is_upper_entire**(*tok_seq*)
True if a sequence is fully upper-cased

educe.rst_dt.learning.features_dev.**is_upper_init**(*tok_seq*)
True if a sequence starts with two upper-cased tokens

educe.rst_dt.learning.features_dev.**product_features**(*feats_g*, *feats_d*, *feats_gd*)
Generate features by taking the product of features.

Parameters

- **feats_g** (*dict(feat_name, feat_val)*) – features of the gov EDU
- **feats_d** (*dict(feat_name, feat_val)*) – features of the dep EDU
- **feats_gd** (*dict(feat_name, feat_val)*) – features of the (gov, dep) edge

Returns **pf** – product features

Return type dict(feat_name, feat_val)

educe.rst_dt.learning.features_dev.**split_feature_space**(*feats_g*, *feats_d*, *feats_gd*, *keep_original=False*, *split_criterion='dir'*)
Split feature space on a criterion.

Current supported criteria are: * 'dir': directionality of attachment, * 'sent': intra/inter-sentential, * 'dir_sent': directionality + intra/inter-sentential.

Parameters

- **feats_g** (*dict(feat_name, feat_val)*) – features of the gov EDU
- **feats_d** (*dict(feat_name, feat_val)*) – features of the dep EDU
- **feats_gd** (*dict(feat_name, feat_val)*) – features of the (gov, dep) edge
- **keep_original** (*boolean, default=False*) – whether to keep or replace the original features with the derived split features
- **split_criterion** (*string*) – feature(s) on which to split the feature space, options are 'dir' for directionality of attachment, 'sent' for intra/inter sentential, 'dir_sent' for their conjunction

Returns **feats_g, feats_d, feats_gd** – dicts of features with their copies

Return type (dict(feat_name, feat_val))

Notes

This function should probably be generalized and moved to a more relevant place.

educe.rst_dt.learning.features_dev.**token_filter_li2014**(*token*)
Token filter defined in Li et al.'s parser.

This filter only applies to tagged tokens.

educe.rst_dt.learning.features_li2014 module

Partial re-implementation of the feature extraction procedure used in *[li2014text]* for discourse dependency parsing on the RST-DT corpus.

Text-level discourse dependency parsing. In Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Vol. 1, pp. 25-35). http://www.aclweb.org/anthology/P/P14/P14-1003.pdf

educe.rst_dt.learning.features_li2014.**build_doc_preprocessor**()
Build the preprocessor for feature extraction in each EDU of doc

educe.rst_dt.learning.features_li2014.**build_edu_feature_extractor**()
Build the feature extractor for single EDUs

educe.rst_dt.learning.features_li2014.**build_pair_feature_extractor**()
Build the feature extractor for pairs of EDUs

TODO: properly emit features on single EDUs ; they are already stored in sf_cache, but under (slightly) different names

educe.rst_dt.learning.features_li2014.**combine_features**(*feats_g*, *feats_d*, *feats_gd*)
Generate features by taking a (linear) combination of features.

I suspect these do not have a great impact, if any, on results.

Parameters

- **feats_g** (*dict(feat_name, feat_val)*) – features of the gov EDU
- **feats_d** (*dict(feat_name, feat_val)*) – features of the dep EDU
- **feats_gd** (*dict(feat_name, feat_val)*) – features of the (gov, dep) edge

Returns **cf** – combined features

Return type dict(feat_name, feat_val)

educe.rst_dt.learning.features_li2014.**extract_pair_length**(*doc*, *edu_info1*, *edu_info2*)
Sentence tuple features

educe.rst_dt.learning.features_li2014.**extract_pair_para**(*doc*, *edu_info1*, *edu_info2*)
Paragraph tuple features

educe.rst_dt.learning.features_li2014.**extract_pair_pos**(*doc*, *edu_info1*, *edu_info2*)
POS tuple features

educe.rst_dt.learning.features_li2014.**extract_pair_sent**(*doc*, *edu_info1*, *edu_info2*)
Sentence tuple features

educe.rst_dt.learning.features_li2014.**extract_pair_word**(*doc*, *edu_info1*, *edu_info2*)
word tuple features

educe.rst_dt.learning.features_li2014.**extract_single_length**(*doc*, *edu_info*, *para_info*)
Sentence features for the EDU

educe.rst_dt.learning.features_li2014.**extract_single_para**(*doc*, *edu_info*, *para_info*)
paragraph features for the EDU

educe.rst_dt.learning.features_li2014.**extract_single_pos**(*doc*, *edu_info*, *para_info*)
POS features for the EDU

educe.rst_dt.learning.features_li2014.**extract_single_sentence**(*doc*, *edu_info*, *para_info*)
Sentence features for the EDU

educe.rst_dt.learning.features_li2014.**extract_single_syntax**(*doc*, *edu_info*, *para_info*)
syntactic features for the EDU

educe.rst_dt.learning.features_li2014.**extract_single_word**(*doc*, *edu_info*, *para_info*)
word features for the EDU

educe.rst_dt.learning.features_li2014.**get_syntactic_labels**(*doc*, *edu_info*)
Syntactic labels for this EDU

educe.rst_dt.learning.features_li2014.**product_features**(*feats_g*, *feats_d*, *feats_gd*)
Generate features by taking the product of features.

Parameters

- **feats_g** (*dict(feat_name, feat_val)*) – features of the gov EDU
- **feats_d** (*dict(feat_name, feat_val)*) – features of the dep EDU
- **feats_gd** (*dict(feat_name, feat_val)*) – features of the (gov, dep) edge

Returns **pf** – product features

Return type dict(feat_name, feat_val)

educe.rst_dt.learning.features_li2014.**token_filter_li2014**(*token*)
Token filter defined in Li et al.'s parser.

This filter only applies to tagged tokens.

educe.rst_dt.util package

Submodules

educe.rst_dt.util.args module

Command line options

educe.rst_dt.util.args.**add_usual_input_args**(*parser*)
Augment a subcommand argparser with typical input arguments. Sometimes your subcommand may require slightly different output arguments, in which case, just don't call this function.

:param doc_subdoc_required force user to supply –doc/–subdoc for this subcommand

:type doc_subdoc_required bool

:param help_suffix appended to –doc/–subdoc help strings :type help_suffix string

`educe.rst_dt.util.args.`**`add_usual_output_args`**(*parser*)
Augment a subcommand argparser with typical output arguments, Sometimes your subcommand may require slightly different output arguments, in which case, just don't call this function.

`educe.rst_dt.util.args.`**`announce_output_dir`**(*output_dir*)
Tell the user where we saved the output

`educe.rst_dt.util.args.`**`get_output_dir`**(*args*)
Return the output directory specified on (or inferred from) the command line arguments, *creating it if necessary*.

We try the following in order:

1. If *–output* is given explicitly, we'll just use/create that

2. OK just make a temporary directory. Later on, you'll probably want to call *announce_output_dir*.

`educe.rst_dt.util.args.`**`read_corpus`**(*args*, *verbose=True*)
Read the section of the corpus specified in the command line arguments.

Submodules

educe.rst_dt.annotation module

Educe-style representation for RST discourse treebank trees

class `educe.rst_dt.annotation.`**`EDU`**(*num*, *span*, *text*, *context=None*, *origin=None*)
Bases: *educe.annotation.Standoff*

An RST leaf node

`context` = None
See the *RSTContext* object

`identifier`()
A global identifier (assuming the origin can be used to uniquely identify an RST tree)

`is_left_padding`()
Returns True for left padding EDUs

classmethod `left_padding`(*context=None*, *origin=None*)
Return a left padding EDU

`num` = None
EDU number (as used in tree node *edu_span*)

`raw_text` = None
text that was in the EDU annotation itself

This is not the same as the text that was in the annotated document, on which all standoff annotations and spans are based.

`set_context`(*context*)
Update the context of this annotation.

`set_origin`(*origin*)
Update the origin of this annotation and any contained within

Parameters **`origin`** (FileId) – File identifier of the origin of this annotation.

span = None
text span

text()
Return the text associated with this EDU. We try to return the underlying annotated text if we have the necessary context; if we not, we just fall back to the raw EDU text

class educe.rst_dt.annotation.**Node**(*nuclearity*, *edu_span*, *span*, *rel*, *context=None*)
Bases: object

A node in an *RSTTree* or *SimpleRSTTree*.

context = None
See the *RSTContext* object

edu_span = None
pair of integers denoting edu span by count

is_nucleus()
A node can either be a nucleus, a satellite, or a root node. It may be easier to work with SimpleRSTTree, in which nodes can only either be nucleus/satellite or much more rarely, root.

is_satellite()
A node can either be a nucleus, a satellite, or a root node.

nuclearity = None
one of Nucleus, Satellite, Root

rel = None
relation label (see *SimpleRSTTree* for a note on the different interpretation of *rel* with this and *RSTTree*)

span = None
span

class educe.rst_dt.annotation.**RSTContext**(*text*, *sentences*, *paragraphs*)
Bases: object

Additional annotations or contextual information that could accompany a RST tree proper. The idea is to have each subtree pointing back to the same context object for easy retrieval.

paragraphs = None
Paragraph annotations pointing back to the text

sentences = None
sentence annotations pointing back to the text

text(*span=None*)
Return the text associated with these annotations (or None), optionally limited to a span

class educe.rst_dt.annotation.**RSTTree**(*node*, *children*, *origin=None*, *verbose=False*)
Bases: *educe.external.parser.SearchableTree*, *educe.annotation.Standoff*

Representation of RST trees which sticks fairly closely to the raw RST discourse treebank one.

edu_span()
Return the span of the tree in terms of EDU count See *self.span* refers more to the character offsets

get_spans(*subtree_filter=None*, *exclude_root=False*, *span_type='edus'*)
Get the spans of a constituency tree.

Each span is described by a triplet (edu_span, nuclearity, relation).

Parameters

- **subtree_filter** (*function, defaults to None*) – Function to filter all local trees.
- **exclude_root** (*boolean, defaults to False*) – If True, exclude the span of the root node. This cannot be expressed with *subtree_filter* because the latter is limited to properties local to each subtree in isolation. Or maybe I just missed something.
- **span_type** (*one of {'edus', 'chars'}*) – Whether each span is expressed on EDU or character indices. Character indices are useful to compare spans from trees whose EDU segmentation differs.

Returns **spans** – List of tuples, each describing a span with a tuple ((edu_start, edu_end), nuclearity, relation).

Return type list of tuple((int, int), str, str)

set_origin(*origin*)
Update the origin of this annotation and any contained within

Parameters **origin** (FileId) – File identifier of the origin of this annotation.

text()
Return the text corresponding to this RST subtree. If the context is set, we return the appropriate segment from the subset of the text. If not we just concatenate the raw text of all EDU leaves.

text_span()

to_pdf(*filename*)
Image representation in PDF.

to_ps(*filename*)
Export as a PostScript image.

This function is used by *_repr_png_*.

exception educe.rst_dt.annotation.**RSTTreeException**(*msg*)
Bases: exceptions.Exception

Exceptions related to RST trees not looking like we would expect them to

class educe.rst_dt.annotation.**SimpleRSTTree**(*node*, *children*, *origin=None*)
Bases: *educe.external.parser.SearchableTree*, *educe.annotation.Standoff*

Possibly easier representation of RST trees to work with:

- binary
- relation labels on parent nodes instead of children

Note that *RSTTree* and *SimpleRSTTree* share the same *Node* type but because of the subtle difference in interpretation you should be extremely careful not to mix and match.

classmethod **from_rst_tree**(*tree*)
Build and return a *SimpleRSTTree* from an *RSTTree*

get_spans(*subtree_filter=None*, *exclude_root=False*, *span_type='edus'*)
Get the spans of a constituency tree.

Each span is described by a triplet (edu_span, nuclearity, relation).

Parameters

- **subtree_filter** (*function, defaults to None*) – Function to filter all local trees.

- **exclude_root** (*boolean, defaults to False*) – If True, exclude the span of the root node. This cannot be expressed with *subtree_filter* because the latter is limited to properties local to each subtree in isolation. Or maybe I just missed something.
- **span_type** (*one of {'edus', 'chars'}*) – Whether each span is expressed on EDU or character indices. Character indices are useful to compare spans from trees whose EDU segmentation differs.

Returns **spans** – List of tuples, each describing a span with a tuple ((edu_start, edu_end), nuclearity, relation).

Return type list of tuple((int, int), str, str)

classmethod incorporate_nuclearity_into_label(*tree*)

Integrate nuclearity of the children into each node's label.

Nuclearity of the children is incorporated in one of two forms, NN for multi- and NS for mono-nuclear relations.

Parameters **tree** (SimpleRSTTree) – The tree of which we want a version with nuclearity incorporated

Returns **mod_tree** – The same tree but with the type of nuclearity incorporated

Return type *SimpleRSTTree*

Note: This is probably not the best way to provide this functionality. In other words, refactoring is much needed here.

---

set_origin(*origin*)

Recursively update the origin for this annotation, ie. a little link to the document metadata for this annotation.

Parameters **origin** (FileId) – File identifier of the origin of this annotation.

text_span()

classmethod to_binary_rst_tree(*tree*, *rel='—'*, *nuc='Root'*)

Build and return a binary *RSTTree* from a *SimpleRSTTree*.

This function is recursive, it essentially pushes the relation label from the parent to the satellite child (for mononuclear relations) or to all nucleus children (for multinuclear relations).

Parameters

- **tree** (SimpleRSTTree) – SimpleRSTTree to convert
- **rel** (*string, optional*) – Relation for the root node of the output
- **nuc** (*string, optional*) – Nuclearity for the root node of the output

Returns **rtree** – The (binary) RSTTree that corresponds to the given SimpleRSTTree

Return type *RSTTree*

educe.rst_dt.annotation.**is_binary**(*tree*)

True if the given RST tree or SimpleRSTTree is indeed binary

educe.rst_dt.corpus module

Corpus management (re-exported by educe.rst_dt)

class educe.rst_dt.corpus.**Reader**(*corpusdir*)
Bases: *educe.corpus.Reader*

See *educe.corpus.Reader* for details

files(*doc_glob=None*)

Parameters **doc_glob** (*str, optional*) – Glob for document names, ie. file basenames. A common pattern is *doc_glob='wsj_*'* to exclude documents whose file basenames are of the form *fileX*. *fileX* documents are damaged compared to *wsj_XX* documents ie. their text and that of the corresponding document in the PTB mismatch, and text formatting is scrambled. For example, the figures reported in the paper of (Li et al., 2014) indicate they only consider *wsj_XX* files.

slurp_subcorpus(*cfiles*, *verbose=False*)
See *educe.rst_dt.parse* for a description of *RSTTree*

class educe.rst_dt.corpus.**RstDtParser**(*corpus_dir*, *args*, *coarse_rels=False*, *fix_pseudo_rels=False*, *nary_enc='chain'*, *nuc_in_label=False*, *exclude_file_docs=False*)
Bases: object

Fake parser that gets annotation from the RST-DT.

Parameters

- **corpus_dir** (*string*) – TODO
- **args** (*TODO*) – TODO
- **coarse_rels** (*boolean, optional*) – If True, relation labels are converted to their coarse-grained equivalent.
- **nary_enc** (*string, optional*) – Conversion method from constituency to dependency tree, for n-ary spans, n > 2, whose kids are all nuclei: 'tree' picks the leftmost nucleus as the head of all the others (effectively a tree), 'chain' attaches each nucleus to its predecessor (effectively a chain).
- **nuc_in_label** (*boolean, optional*) – If True, incorporate nuclearity into the label (ex: elaboration-NS) ; currently BROKEN (defined on SimpleRSTTree only).
- **exclude_file_docs** (*boolean, default False*) – If True, ignore fileX files.

decode(*doc_key*)
Decode a document from the RST-DT (gold)

Parameters **doc_key** (*string ?*) – Identifier (in corpus) of the document we want to decode.

Returns **doc** – Bunch of information about this document notably its list of EDUs and the structures defined on them: RSTTree, SimpleRSTTree, RstDepTree.

Return type *DocumentPlus*

parse(*doc*)
Parse the document using the RST-DT (gold).

segment(*doc*)
Segment the document into EDUs using the RST-DT (gold).

class educe.rst_dt.corpus.**RstRelationConverter**(*relmap_file*)
Bases: object

Converter for RST relations (labels)

Known to work on RstTree, possibly SimpleRstTree (untested).

convert_dtree(*dtree*)
Change relation labels in an RstDepTree using the label mapping.

See attribute *self.convert_label*.

Parameters **dtree** (RstDepTree) – RST dtree

Returns **dtree** – RST dtree with mapped labels.

Return type *RstDepTree*

convert_label(*label*)
Convert a label following the mapping, lowercased otherwise

convert_tree(*rst_tree*)
Change relation labels in rst_tree using the mapping

educe.rst_dt.corpus.**id_to_path**(*k*)
Given a fleshed out FileId (none of the fields are None), return a filepath for it following RST Discourse Treebank conventions.

You will likely want to add your own filename extensions to this path

educe.rst_dt.corpus.**mk_key**(*doc*)
Return an corpus key for a given document name

educe.rst_dt.deptree module

Convert RST trees to dependency trees and back.

class educe.rst_dt.deptree.**RstDepTree**(*edus=[]*, *origin=None*, *nary_enc='chain'*)
Bases: object

RST dependency tree

edus
list of EDU – List of the EDUs of this document.

origin
Document?, optional – TODO

nary_enc
one of {'chain', 'tree'}, optional – Type of encoding used for n-ary relations: 'chain' or 'tree'. This determines for example how fragmented EDUs are resolved.

add_dependencies(*gov_num*, *dep_nums*, *labels=None*, *nucs=None*, *rank=None*)
Add a set of dependencies with a unique governor and rank.

Parameters

- **gov_num** (*int*) – Number of the head EDU
- **dep_nums** (*list of int*) – Number of the modifier EDUs
- **labels** (*list of string, optional*) – Labels of the dependencies
- **nuc** (*list of string, one of [NUC_S, NUC_N]*) – Nuclearity of the modifiers
- **rank** (*integer, optional*) – Rank of the modifiers in the order of attachment to the head. *None* means it is not given declaratively and it is instead inferred from the rank of modifiers previously attached to the head.

add_dependency(*gov_num*, *dep_num*, *label=None*, *nuc='Satellite'*, *rank=None*)
Add a dependency between two EDUs.

Parameters

- **gov_num** (*int*) – Number of the head EDU
- **dep_num** (*int*) – Number of the modifier EDU
- **label** (*string, optional*) – Label of the dependency
- **nuc** (*string, one of [NUC_S, NUC_N]*) – Nuclearity of the modifier
- **rank** (*integer, optional*) – Rank of the modifier in the order of attachment to the head. *None* means it is not given declaratively and it is instead inferred from the rank of modifiers previously attached to the head.

append_edu(*edu*)
Append an EDU to the list of EDUs

deps(*gov_idx*)
Get the ordered list of dependents of an EDU

fragmented_edus()
Get the fragmented EDUs in this RST tree.

Fragmented EDUs are made of two or more EDUs linked by "same-unit" relations.

Returns **frag_edus** – Each fragmented EDU is given as a tuple of the indices of the fragments.

Return type list of tuple of int

classmethod from_rst_tree(*rtree*, *nary_enc='tree'*)
Converts an RSTTree‘ to an *RstDepTree*.

Parameters **nary_enc** (*one of {'chain', 'tree'}*) – If ‘chain’, the given RSTTree is binarized first.

classmethod from_simple_rst_tree(*rtree*)
Converts a SimpleRSTTree‘ to an *RstDepTree*

get_dependencies(*lbl_type='rel'*)
Get the list of dependencies in this dependency tree.

Each dependency is a 3-uple (gov, dep, label), gov and dep being EDUs.

Parameters **lbl_type** (*one of {'rel', 'rel+nuc'} (TODO 'rel+nuc+rnk'?)*) – Type of the labels.

real_roots_idx()
Get the list of the indices of the real roots

set_origin(*origin*)
Update the origin of this annotation.

Parameters **origin** (FileId) – File identifier of the origin of this annotation.

set_root(*root_num*)
Designate an EDU as a real root of the RST tree structure

spans()
For each EDU, get the tree span it dominates (on EDUs).

Dominance here is recursively defined.

Returns

- **span_beg** (*array of int*) – Index of the leftmost EDU dominated by an EDU.
- **span_end** (*array of int*) – Index of the rightmost EDU dominated by an EDU.

exception `educe.rst_dt.deptree.`**`RstDtException`**(*msg*)
Bases: `exceptions.Exception`

Exceptions related to conversion between RST and DT trees. The general expectation is that we only raise these on bad input, but in practice, you may see them more in cases of implementation error somewhere in the conversion process.

`educe.rst_dt.deptree.`**`binary_to_nary`**(*nary_enc*, *pairs*)
Retrieve nary relations from a set of binary relations.

Parameters

- **nary_enc** (*one of {"chain", "tree"}*) – Encoding from n-ary to binary relations.
- **pairs** (*iterable of pairs of identifier (ex: integer, string. ..)*) – Binary relations.

Returns **nary_rels** – Nary relations.

Return type list of tuples of identifiers

educe.rst_dt.document_plus module

This submodule implements a document with additional information.

class `educe.rst_dt.document_plus.`**`DocumentPlus`**(*key*, *grouping*, *rst_context*)
Bases: `object`

A document and relevant contextual information

`align_with_doc_structure`()
Align EDUs with the document structure (paragraph and sentence).

Determine which paragraph and sentence (if any) surrounds this EDU. Try to accomodate the occasional off-by-a-smidgen error by folks marking these EDU boundaries, eg. original text:

Para1: “Magazines are not providing us in-depth information on circulation,” said Edgar Bronfman Jr., .. “How do readers feel about the magazine?... Research doesn’t tell us whether people actually do read the magazines they subscribe to.”

Para2: Reuben Mark, chief executive of Colgate-Palmolive, said...

Marked up EDU is wide to the left by three characters: “

Reuben Mark, chief executive of Colgate-Palmolive, said...

`align_with_raw_words`()
Compute for each EDU the raw tokens it contains

This is a dirty temporary hack to enable backwards compatibility. There should be one clean text per document, one tokenization and so on, but, well.

`align_with_tokens`(*verbose=False*)
Compute for each EDU the overlapping tokens.

`align_with_trees`(*strict=False*)
Compute for each EDU the overlapping trees

all_edu_pairs(*ordered=True*)
Generate all EDU pairs of a document.

Parameters **ordered** (*boolean, defaults to True*) – If True, generate all ordered pairs of EDUs, otherwise (half as many) unordered pairs.

Returns **all_pairs** – All pairs of EDUs in this document.

Return type [(*EDU*, *EDU*)]

relations(*du_pairs*, *lbl_type='rel'*, *ordered=True*)
Get the relation that holds in each of the DU pairs.

As of 2016-09-30, this function has a unique caller: doc_vectorizer.DocumentLabelExtractor._extract_labels()
.

Parameters

- **du_pairs** (*[(DU, DU)]*) – List of DU pairs.
- **lbl_type** (*one of {'rel', 'rel+nuc'}*) – Type of label.
- **ordered** (*boolean, defaults to True*) – If True, du_pairs are considered ordered, otherwise the label of either (edu1, edu2) or (edu2, edu1) is returned (if not None).

Returns **erels** – Relation for each pair of DUs.

Return type `list` of `str`

same_unit_candidates()
Generate all EDU pairs that could be a same-unit.

We use the following filters: * right-attachment: i < j, * same sentence: edu2sent[i] == edu2sent[j], * len > 1: i + 1 < j

set_syn_ctrees(*tkd_trees*, *lex_heads=None*)
Set syntactic constituency trees for this document.

Parameters

- **tkd_trees** (*list of nltk.tree.Tree*) – Syntactic constituency trees for this document.
- **lex_heads** (*list of (TODO: see find_lexical_heads), optional*) – List of lexical heads for each node of each tree.

set_tokens(*tokens*)
Set tokens for this document.

Parameters **tokens** (*list of Token*) – List of tokens for this document.

educe.rst_dt.document_plus.**align_edus_with_paragraphs**(*doc_edus*, *doc_paras*, *text*, *strict=False*)
Align EDUs with paragraphs, if any.

Parameters

- **doc_edus** –
- **doc_paras** –
- **strict** –

Returns **edu2para** – Map each EDU to the index of its enclosing paragraph. If an EDU is not properly enclosed in a paragraph, the associated index is None. For files with no paragraph marking (e.g. *fileX* files), returns None.

Return type list(int or None) or None

educe.rst_dt.document_plus.**containing**(*span*)
span -> anno -> bool

if this annotation encloses the given span

educe.rst_dt.graph module

Converter from RST Discourse Treebank trees to educe-style hypergraphs

class educe.rst_dt.graph.**DotGraph**(*anno_graph*)
Bases: *educe.graph.DotGraph*

A dot representation of this graph for visualisation. The *to_string()* method is most likely to be of interest here

class educe.rst_dt.graph.**Graph**
Bases: *educe.graph.Graph*

classmethod from_doc(*corpus*, *doc_key*)

educe.rst_dt.parse module

From RST discourse treebank trees to Educe-style objects (reading the format from Di Eugenio's corpus of instructional texts).

The main classes of interest are *RSTTree* and *EDU*. *RSTTree* can be treated as an NLTK Tree structure. It is also an educe *Standoff* object, which means that it points to other RST trees (their children) or to *EDU*.

educe.rst_dt.parse.**parse_lightweight_tree**(*tstr*)
Parse lightweight RST debug syntax into SimpleRSTTree, eg.

```
(R:attribution
   (N:elaboration (N foo) (S bar)
   (S quux)))
```

This is motly useful for debugging or for knocking out quick examples

educe.rst_dt.parse.**parse_rst_dt_tree**(*tstr*, *context=None*)
Read a single RST tree from its RST DT string representation. If context is set, align the tree with it. You should really try to pass in a context (see *RSTContext* if you can, the None case is really intended for testing, or in cases where you don't have an original text)

educe.rst_dt.parse.**read_annotation_file**(*anno_filename*, *text_filename*)
Read a single RST tree

educe.rst_dt.ptb module

Alignment the RST-WSJ-corpus with the Penn Treebank

class educe.rst_dt.ptb.**PtbParser**(*corpus_dir*)
Bases: object

Gold parser that gets annotations from the PTB.

It uses an instantiated NLTK BracketedParseCorpusReader for the PTB section relevant to the RST DT corpus.

Note that the path you give to this will probably end with something like *parsed/mrg/wsj*

parse(*doc*)
Parse a document, using the gold PTB annotation.

Given a document, return a list of educified PTB parse trees (one per sentence).

These are almost the same as the trees that would be returned by the *parsed_sents* method, except that each leaf/node is associated with a span within the RST DT text.

Note: does nothing if there is no associated PTB corpus entry.

Parameters **doc** (DocumentPlus) – Rich representation of the document.

Returns **doc** – Rich representation of the document, with syntactic constituency trees.

Return type *DocumentPlus*

tokenize(*doc*)
Tokenize the document text using the PTB gold annotation.

Parameters **doc** (DocumentPlus) – Rich representation of the document.

Returns **doc** – Rich representation of the document, with tokenization.

Return type *DocumentPlus*

educe.rst_dt.ptb.**align_edus_with_sentences**(*edus*, *syn_trees*, *strict=False*)
Map each EDU to its sentence.

If an EDU span overlaps with more than one sentence span, the sentence with maximal overlap is chosen.

Parameters

- **edus** (*list (EDU)*) – List of EDUs.
- **syn_trees** (*list (Tree)*) – List of syntactic trees, one per sentence.
- **strict** (*boolean, default False*) – If True, raise an error if an EDU does not map to exactly one sentence.

Returns **edu2sent** – Map from EDU to (0-based) sentence index or None.

Return type list(int or None)

educe.rst_dt.rst_wsj_corpus module

This module provides loaders for file formats found in the RST-WSJ-corpus.

educe.rst_dt.rst_wsj_corpus.**load_rst_wsj_corpus_edus_file**(*f*)
Load a file that contains the EDUs of a document.

Return clean text and the list of EDU offsets on the clean text.

educe.rst_dt.rst_wsj_corpus.**load_rst_wsj_corpus_text_file**(*f*)
Load a text file from the RST-WSJ-CORPUS.

Return the text plus its sentences and paragraphs.

The corpus contains two types of text files, so this function is mainly an entry point that delegates to the appropriate function.

educe.rst_dt.rst_wsj_corpus.**load_rst_wsj_corpus_text_file_file**(*f*)
Load a text file whose name is of the form *file##*

These files do not mark paragraphs. Each line contains a sentence preceded by two or three leading spaces.

educe.rst_dt.rst_wsj_corpus.**load_rst_wsj_corpus_text_file_wsj**(*f*)
Load a text file whose name is of the form *wsj_##*

By convention:

- paragraphs are separated by double newlines
- sentences by single newlines

Note that this segmentation isn't particularly reliable, and seems to both over- (e.g. cut at some abbreviations, like "Prof.") and under-segment (e.g. not separate contiguous sentences). It shouldn't be taken too seriously, but if you need some sort of rough approximation, it may be helpful.

educe.rst_dt.sdrt module

Convert RST trees to SDRT style EDU/CDU annotations.

The core of the conversion is *rst_to_sdrt* which produces an intermediary pointer based representation (a single *CDU* pointing to other CDUs and EDUs).

A fancier variant, *rst_to_glozz_sdrt* wraps around this core and further converts the *CDU* into a Glozz-friendly form

class educe.rst_dt.sdrt.**CDU**(*members*, *rel_insts*)
Complex Discourse Unit.

A CDU contains one or more discourse units, and tracks relation instances between its members. Both CDU and EDU are discourse units.

members
list of Unit or Scheme – Immediate member units (EDUs and CDUs) of this CDU.

rel_insts
list of Relation – Relation instances between immediate members of this CDU.

class educe.rst_dt.sdrt.**RelInst**(*source*, *target*, *type*)
Relation instance.

educe.annotation calls these 'Relation's which is really more in keeping with how Glozz class them, but properly speaking relation instance is a better name.

source
Unit? – Source of the relation instance.

target
Unit? – Target of the relation instance.

type
string – Name of the relation.

educe.rst_dt.sdrt.**debug_du_to_tree**(*m*)
Tree representation of CDU.

The set of relation instances is treated as the parent of each node. Loses information ; should only be used for debugging purposes.

educe.rst_dt.sdrt.**rst_to_glozz_sdrt**(*rst_tree*, *annotator='ldc'*)
From an RST tree to a STAC-like version using Glozz annotations. Uses *rst_to_sdrt*

educe.rst_dt.sdrt.**rst_to_sdrt**(*tree*)
From *RSTTree* to *CDU* or *EDU* (recursive, top-down transformation). We recognise three patterns walking down the tree (anything else is considered to be an error):

- Pre-terminal nodes: Return the leaf EDU

- Mono-nuclear, N satellites: Return a CDU with a relation instance from the nucleus to each satellite. As an informal example, given *X(attribution:S1, N, explanation-argumentative:S2)*, we return a CDU with *sdrt(N) – attribution –> sdrt(S1)* and *sdrt(N) – explanation-argumentative –> sdrt(S2)*
- Multi-nuclear, 0 satellites: Return a CDU with a relation instance across each successive nucleus (assume the same relation). As an informal example, given *X(List:N1, List:N2, List:N3)*, we return a CDU containing *sdrt(N1) –List–> sdrt(N2) – List –> sdrt(N3)*.

educe.rst_dt.text module

Educe-style annotations for RST discourse treebank text objects (paragraphs and sentences)

class `educe.rst_dt.text.`**`Paragraph`**(*num*, *sentences*)
Bases: *educe.annotation.Standoff*

A paragraph is a sequence of 'Sentence's (also standoff annotations).

classmethod `left_padding`(*sentences*)
Return a left padding Paragraph

`num` = **None**
paragraph ID in document

`sentences` = **None**
sentence-level annotations

class `educe.rst_dt.text.`**`Sentence`**(*num*, *span*)
Bases: *educe.annotation.Standoff*

Just a text span really

classmethod `left_padding`()
Return a left padding Sentence

`num` = **None**
sentence ID in document

`text_span`()

`educe.rst_dt.text.`**`clean_edu_text`**(*text*)
Strip metadata from EDU text and compress extraneous whitespace

4.3.6 educe.stac package

Conventions specific to the STAC project

This includes things like

- corpus layout (see *corpus_files*)
- which annotations are of interest
- renaming/deleting/collapsing annotation labels

Subpackages

educe.stac.learning package

Helpers for machine-learning tasks

Submodules

educe.stac.learning.addressee module

EDU addressee prediction

`educe.stac.learning.addressee.`**`guess_addressees_for_edu`**(*contexts*, *players*, *edu*)
return a set of possible addressees for the given EDU or None if unclear

At the moment, the basis for our guesses is very crude: we simply guess that we have an addresee if the EDU ends or starts with their name

`educe.stac.learning.addressee.`**`is_emoticon`**(*token*)
True if the token is tagged as an emoticon

`educe.stac.learning.addressee.`**`is_preposition`**(*token*)
True if the token is tagged as a preposition

`educe.stac.learning.addressee.`**`is_punct`**(*token*)
True if the token is tagged as punctuation

`educe.stac.learning.addressee.`**`is_verb`**(*token*)
True if the token is tagged as a verb

educe.stac.learning.doc_vectorizer module

This submodule implements document vectorizers

class `educe.stac.learning.doc_vectorizer.`**`DialogueActVectorizer`**(*instance_generator*, *labels*)

Bases: `object`

Dialogue act extractor for the STAC corpus.

`transform`(*raw_documents*)
Learn the label encoder and return a vector of labels

There is one label per instance extracted from raw_documents.

Parameters **`raw_documents`** (list of *educe.stac.fusion.Dialogue*) – List of dialogues.

Yields **inst_lbl** (*int*) – (Integer) label for the next instance.

class `educe.stac.learning.doc_vectorizer.`**`LabelVectorizer`**(*instance_generator*, *labels*, *zero=False*)

Bases: `object`

Label extractor for the STAC corpus.

`transform`(*raw_documents*)
Learn the label encoder and return a vector of labels

There is one label per instance extracted from raw_documents.

Parameters **`raw_documents`** (*list of ?*) – Raw documents.

Yields **inst_lbl** (*int*) – (Integer) label for the next instance.

educe.stac.learning.features module

Feature extraction library functions for STAC corpora. The feature extraction script (rel-info) is a lightweight frontend to this library

exception `educe.stac.learning.features.`**`CorpusConsistencyException`**(*msg*)
Bases: `exceptions.Exception`

Exceptions which arise if one of our expecations about the corpus data is violated, in short, weird things we don't know how to handle. We should avoid using this for things which are definitely bugs in the code, and not just weird things in the corpus we didn't know how to handle.

class `educe.stac.learning.features.`**`DocEnv`**(*inputs*, *current*, *sf_cache*)
Bases: `tuple`

`current`
Alias for field number 1

`inputs`
Alias for field number 0

`sf_cache`
Alias for field number 2

class `educe.stac.learning.features.`**`DocumentPlus`**(*key*, *doc*, *unitdoc*, *players*, *parses*)
Bases: `tuple`

`doc`
Alias for field number 1

`key`
Alias for field number 0

`parses`
Alias for field number 4

`players`
Alias for field number 3

`unitdoc`
Alias for field number 2

class `educe.stac.learning.features.`**`EduGap`**(*sf_cache*, *inner_edus*, *turns_between*)
Bases: `tuple`

`inner_edus`
Alias for field number 1

`sf_cache`
Alias for field number 0

`turns_between`
Alias for field number 2

class `educe.stac.learning.features.`**`FeatureCache`**(*inputs*, *current*)
Bases: `dict`

Cache for single edu features. Retrieving an item from the cache lazily computes/memoises the single EDU features for it.

`expire`(*edu*)
Remove an edu from the cache if it's in there

class educe.stac.learning.features.**FeatureInput**(*corpus*, *postags*, *parses*, *lexicons*, *pdtb_lex*, *verbnet_entries*, *inquirer_lex*)

Bases: tuple

corpus
Alias for field number 0

inquirer_lex
Alias for field number 6

lexicons
Alias for field number 3

parses
Alias for field number 2

pdtb_lex
Alias for field number 4

postags
Alias for field number 1

verbnet_entries
Alias for field number 5

class educe.stac.learning.features.**InquirerLexKeyGroup**(*lexicon*)

Bases: *educe.learning.keys.KeyGroup*

One feature per Inquirer lexicon class

fill(*current*, *edu*, *target=None*)
See *SingleEduSubgroup*

classmethod key_prefix()
All feature keys in this lexicon should start with this string

mk_field(*entry*)
From verb class to feature key

mk_fields()
Feature name for each relation in the lexicon

class educe.stac.learning.features.**LexKeyGroup**(*lexicon*)

Bases: *educe.learning.keys.KeyGroup*

The idea here is to provide a feature per lexical class in the lexicon entry

fill(*current*, *edu*, *target=None*)
See *SingleEduSubgroup*

key_prefix()
Common CSV header name prefix to all columns based on this particular lexicon

mk_field(*cname*, *subclass=None*)
For a given lexical class, return the name of its feature in the CSV file

mk_fields()
CSV field names for each entry/class in the lexicon

class educe.stac.learning.features.**LexWrapper**(*key*, *filename*, *classes*)

Bases: object

Configuration options for a given lexicon: where to find it, what to call it, what sorts of results to return

read(*lexdir*)
Read and store the lexicon as a mapping from words to their classes

class educe.stac.learning.features.**MergedLexKeyGroup**(*inputs*)
Bases: *educe.learning.keys.MergedKeyGroup*

Single-EDU features based on lexical lookup.

fill(*current*, *edu*, *target=None*)
See *SingleEduSubgroup*

class educe.stac.learning.features.**PairKeys**(*inputs*, *sf_cache=None*)
Bases: *educe.learning.keys.MergedKeyGroup*

Features for pairs of EDUs

fill(*current*, *edu1*, *edu2*, *target=None*)
See *PairSubgroup*

one_hot_values_gen(*suffix=''*)

class educe.stac.learning.features.**PairSubgroup**(*description*, *keys*)
Bases: *educe.learning.keys.KeyGroup*

Abstract keygroup for subgroups of the merged PairKeys. We use these subgroup classes to help provide modularity, to capture the idea that the bits of code that define a set of related feature vector keys should go with the bits of code that also fill them out

fill(*current*, *edu1*, *edu2*, *target=None*)
Fill out a vector's features (if the vector is None, then we just fill out this group; but in the case of a merged key group, you may find it desirable to fill out the merged group instead)

class educe.stac.learning.features.**PairSubgroup_Gap**(*sf_cache*)
Bases: *educe.stac.learning.features.PairSubgroup*

Features related to the combined surrounding context of the two EDUs

fill(*current*, *edu1*, *edu2*, *target=None*)

class educe.stac.learning.features.**PairSubgroup_Tuple**(*inputs*, *sf_cache*)
Bases: *educe.stac.learning.features.PairSubgroup*

artificial tuple features

fill(*current*, *edu1*, *edu2*, *target=None*)

class educe.stac.learning.features.**PdtbLexKeyGroup**(*lexicon*)
Bases: *educe.learning.keys.KeyGroup*

One feature per PDTB marker lexicon class

fill(*current*, *edu*, *target=None*)
See *SingleEduSubgroup*

classmethod key_prefix()
All feature keys in this lexicon should start with this string

mk_field(*rel*)
From relation name to feature key

mk_fields()
Feature name for each relation in the lexicon

class educe.stac.learning.features.**SingleEduKeys**(*inputs*)
Bases: *educe.learning.keys.MergedKeyGroup*

Features for a single EDU

fill(*current*, *edu*, *target=None*)
See *SingleEduSubgroup.fill*

class educe.stac.learning.features.**SingleEduSubgroup**(*description*, *keys*)
Bases: *educe.learning.keys.KeyGroup*

Abstract keygroup for subgroups of the merged SingleEduKeys. We use these subgroup classes to help provide modularity, to capture the idea that the bits of code that define a set of related feature vector keys should go with the bits of code that also fill them out

fill(*current*, *edu*, *target=None*)
Fill out a vector's features (if the vector is None, then we just fill out this group; but in the case of a merged key group, you may find it desirable to fill out the merged group instead)

This defaults to _magic_fill if you don't implement it.

class educe.stac.learning.features.**SingleEduSubgroup_Chat**
Bases: *educe.stac.learning.features.SingleEduSubgroup*

Single-EDU features based on the EDU's relationship with the chat structure (eg turns, dialogues).

class educe.stac.learning.features.**SingleEduSubgroup_Parser**
Bases: *educe.stac.learning.features.SingleEduSubgroup*

Single-EDU features that come out of a syntactic parser.

class educe.stac.learning.features.**SingleEduSubgroup_Punct**
Bases: *educe.stac.learning.features.SingleEduSubgroup*

punctuation features

class educe.stac.learning.features.**SingleEduSubgroup_Token**
Bases: *educe.stac.learning.features.SingleEduSubgroup*

word/token-based features

class educe.stac.learning.features.**VerbNetEntry**(*classname*, *lemmas*)
Bases: tuple

classname
Alias for field number 0

lemmas
Alias for field number 1

class educe.stac.learning.features.**VerbNetLexKeyGroup**(*ventries*)
Bases: *educe.learning.keys.KeyGroup*

One feature per VerbNet lexicon class

fill(*current*, *edu*, *target=None*)
See *SingleEduSubgroup*

classmethod key_prefix()
All feature keys in this lexicon should start with this string

mk_field(*ventry*)
From verb class to feature key

mk_fields()
Feature name for each relation in the lexicon

educe.stac.learning.features.**clean_chat_word**(*token*)
Given a word and its postag (educe PosTag representation) return a somewhat tidied up version of the word.

- Sequences of the same letter greater than length 3 are shortened to just length three
- Letter is lower cased

educe.stac.learning.features.**clean_dialogue_act**(*act*)
Knock out temporary markers used during corpus annotation

educe.stac.learning.features.**dialogue_act_pairs**(*current*, *cache*, *edu1*, *edu2*)
tuple of dialogue acts for both EDUs

educe.stac.learning.features.**edu_position_in_turn**(_, *edu*)
relative position of the EDU in the turn

educe.stac.learning.features.**edu_text_feature**(*wrapped*)
Lift a text based feature into a standard single EDU one

```
(String -> a) ->
((Current, Edu) -> a)
```

educe.stac.learning.features.**emoticons**(*tokens*)
Given some tokens, return just those which are emoticons

educe.stac.learning.features.**enclosed_lemmas**(*span*, *parses*)
Given a span and a list of parses, return any lemmas that are within that span

educe.stac.learning.features.**enclosed_trees**(*span*, *trees*)
Return the biggest (sub)trees in xs that are enclosed in the span

educe.stac.learning.features.**ends_with_bang**(*current*, *edu*)
if the EDU text ends with '!'

educe.stac.learning.features.**ends_with_qmark**(*current*, *edu*)
if the EDU text ends with '?'

educe.stac.learning.features.**extract_pair_features**(*inputs*, *stage*)
Extraction for all relevant pairs in a document (generator)

educe.stac.learning.features.**extract_single_features**(*inputs*, *stage*)
Return a dictionary for each EDU

educe.stac.learning.features.**feat_annotator**(*current*, *edu1*, *edu2*)
annotator for the subdoc

educe.stac.learning.features.**feat_end**(_, *edu*)
text span end

educe.stac.learning.features.**feat_has_emoticons**(_, *edu*)
if the EDU has emoticon-tagged tokens

educe.stac.learning.features.**feat_id**(_, *edu*)
some sort of unique identifier for the EDU

educe.stac.learning.features.**feat_is_emoticon_only**(_, *edu*)
if the EDU consists solely of an emoticon

educe.stac.learning.features.**feat_start**(_, *edu*)
text span start

educe.stac.learning.features.**get_players**(*inputs*)
Return a dictionary mapping each document to the set of players in that document

educe.stac.learning.features.**has_FOR_np**(*current*, *edu*)
if the EDU has the pattern IN(for).. NP

educe.stac.learning.features.**has_correction_star**(*current*, *edu*)
if the EDU begins with a '*' but does not contain others

educe.stac.learning.features.**has_inner_question**(*current*, *gap*, *_edu1*, *_edu2*)
if there is an intervening EDU that is a question

educe.stac.learning.features.**has_one_of_words**(*sought*, *tokens*, *norm=<function <lambda>>*)
Given a set of words, a collection tokens, return True if the tokens contain words match one of the desired words, modulo some minor normalisations like lowercasing.

educe.stac.learning.features.**has_pdtb_markers**(*markers*, *tokens*)
Given a sequence of tagged tokens, return True if any of the given PDTB markers appears within the tokens

educe.stac.learning.features.**has_player_name_exact**(*current*, *edu*)
if the EDU text has a player name in it

educe.stac.learning.features.**has_player_name_fuzzy**(*current*, *edu*)
if the EDU has a word that sounds like a player name

educe.stac.learning.features.**is_just_emoticon**(*tokens*)
Return true if a sequence of tokens consists of a single emoticon

educe.stac.learning.features.**is_nplike**(*anno*)
is some sort of NP annotation from a parser

educe.stac.learning.features.**is_question**(*current*, *edu*)
if the EDU is (or contains) a question

educe.stac.learning.features.**is_question_pairs**(*current*, *cache*, *edu1*, *edu2*)
boolean tuple: if each EDU is a question

educe.stac.learning.features.**lemma_subject**(**args*, ***kwargs*)
the lemma corresponding to the subject of this EDU

educe.stac.learning.features.**lexical_markers**(*lclass*, *tokens*)
Given a dictionary (words to categories) and a text span, return all the categories of words that appear in that set.

Note that for now we are doing our own white-space based tokenisation, but it could make sense to use a different source of tokens instead

educe.stac.learning.features.**map_topdown**(*good*, *prunable*, *trees*)
Do topdown search on all these trees, concatenate results.

educe.stac.learning.features.**mk_env**(*inputs*, *people*, *key*)
Pre-process and bundle up a representation of the current document

educe.stac.learning.features.**mk_envs**(*inputs*, *stage*)
Generate an environment for each document in the corpus within the given stage.

The environment pools together all the information we have on a single document

educe.stac.learning.features.**mk_high_level_dialogues**(*inputs*, *stage*)
Generate all relevant EDU pairs for a document (generator)

educe.stac.learning.features.**mk_is_interesting**(*args*, *single*)
Return a function that filters corpus keys to pick out the ones we specified on the command line

We have two cases here: for pair extraction, we just want to grab the units and if possible the discourse stage. In live mode, there won't be a discourse stage, but that's fine because we can just fall back on units.

For single extraction (dialogue acts), we'll also want to grab the units stage and fall back to unannotated when in live mode. This is made a bit trickier by the fact that unannotated does not have an annotator, so we have to accomodate that.

Phew.

It's a bit specific to feature extraction in that here we are trying

educe.stac.learning.features.**num_edus_between**(*_current*, *gap*, *_edu1*, *_edu2*)
number of intervening EDUs (0 if adjacent)

educe.stac.learning.features.**num_nonling_tstars_between**(*_current*, *gap*, *_edu1*, *_edu2*)
number of non-linguistic turn-stars between EDUs

educe.stac.learning.features.**num_speakers_between**(*_current*, *gap*, *_edu1*, *_edu2*)
number of distinct speakers in intervening EDUs

educe.stac.learning.features.**num_tokens**(*_*, *edu*)
length of this EDU in tokens

educe.stac.learning.features.**player_addresees**(*edu*)
The set of people spoken to during an edu annotation. This excludes known non-players, like 'All', or '?', or 'Please choose...',

educe.stac.learning.features.**players_for_doc**(*corpus*, *kdoc*)
Return the set of speakers/addressees associated with a document.

In STAC, documents are semi-arbitrarily cut into sub-documents for technical and possibly ergonomic reasons, ie. meaningless as far as we are concerned. So to find all speakers, we would have to search all the subdocuments of a single document.

```
(Corpus, String) -> Set String
```

educe.stac.learning.features.**position_in_dialogue**(*_*, *edu*)
relative position of the turn in the dialogue

educe.stac.learning.features.**position_in_game**(*_*, *edu*)
relative position of the turn in the game

educe.stac.learning.features.**position_of_speaker_first_turn**(*edu*)
Given an EDU context, determine the position of the first turn by that EDU's speaker relative to other turns in that dialogue.

educe.stac.learning.features.**read_corpus_inputs**(*args*)
Read and filter the part of the corpus we want features for

educe.stac.learning.features.**read_pdtb_lexicon**(*args*)
Read and return the local PDTB discourse marker lexicon.

educe.stac.learning.features.**real_dialogue_act**(*edu*)
Given an EDU in the 'discourse' stage of the corpus, return its dialogue act from the 'units' stage

educe.stac.learning.features.**relation_dict**(*doc*, *quiet=False*)
Return the relations instances from a document in the form of an id pair to label dictionary

If there is more than one relation between a pair of EDUs we pick one of them arbitrarily and ignore the other

educe.stac.learning.features.**same_speaker**(*current*, *_*, *edu1*, *edu2*)
if both EDUs have the same speaker

educe.stac.learning.features.**same_turn**(*current*, *_*, *edu1*, *edu2*)
if both EDUs are in the same turn

educe.stac.learning.features.**speaker_already_spoken_in_dialogue**(*_*, *edu*)
if the speaker for this EDU is the same as that of a previous turn in the dialogue

educe.stac.learning.features.**speaker_id**(*_*, *edu*)
Get the speaker ID

educe.stac.learning.features.**speaker_started_the_dialogue**(*_*, *edu*)
if the speaker for this EDU is the same as that of the first turn in the dialogue

educe.stac.learning.features.**speakers_first_turn_in_dialogue**(*_*, *edu*)
position in the dialogue of the turn in which the speaker for this EDU first spoke

educe.stac.learning.features.**strip_cdus**(*corpus*, *mode*)
For all documents in a corpus, remove any CDUs and relink the document according to the desired mode. This mutates the corpus.

educe.stac.learning.features.**subject_lemmas**(*span*, *trees*)
Given a span and a list of dependency trees, return any lemmas which are marked as being some subject in that span

educe.stac.learning.features.**turn_follows_gap**(*_*, *edu*)
if the EDU turn number is > 1 + previous turn

educe.stac.learning.features.**type_text**(*wrapped*)
Given a feature that emits text, clean its output up so to work with a wide variety of csv parsers

```
(a -> String) ->
(a -> String)
```

educe.stac.learning.features.**word_first**(**args*, ***kwargs*)
the first word in this EDU

educe.stac.learning.features.**word_last**(**args*, ***kwargs*)
the last word in this EDU

educe.stac.lexicon package

Submodules

educe.stac.lexicon.markers module

API on discourse markers (lexicon I/O mostly)

class educe.stac.lexicon.markers.**LexConn**(*infile*, *version='2'*, *stop=set([u'xe0', u'ou', u'en', u'pour', u'et'])*)

get_by_form(*form*)

get_by_id(*id*)

get_by_lemma(*lemma*)

class educe.stac.lexicon.markers.**Marker**(*elmt*, *version='2'*, *stop=set([u'xe0', u'ou', u'en', u'pour', u'et'])*)
wrapper class for discourse marker read from Lexconn, version 1 or 2

should include at least id, cat (grammatical category) version 1 has type (coord/subord) version 2 has grammatical host and lemma

get_forms()

get_lemma()

get_relations()

educe.stac.lexicon.pdtb_markers module

Lexicon of discourse markers.

Cheap and cheerful phrasal lexicon format used in the STAC project. Maps sequences of multiword expressions to relations they mark

as ; explanation explanation* background as a result ; result result* for example ; elaboration if:then ; conditional on the one hand:on the other hand

One entry per line. Sometimes you have split expressions, like "on the one hand X, on the other hand Y" (we model this by saying that we are working with sequences of expressions, rather than single expressions). Phrases can be associated with 0 to N relations (interpreted as disjunction; if *wedge* appears (LaTeX for the "logical and" operator), it is ignored).

class educe.stac.lexicon.pdtb_markers.**Marker**(*exprs*)
Bases: object

A marker here is a sort of template consisting of multiword expressions and holes, eg. "on the one hand, XXX, on the other hand YYY". We represent this is as a sequence of Multiword

classmethod any_appears_in(*markers*, *words*, *sep='#####'*)
Return True if any of the given markers appears in the word sequence.

See *appears_in* for details.

appears_in(*words*, *sep='#####'*)
Given a sequence of words, return True if this marker appears in that sequence.

We use a *very* liberal defintion here. In particular, if the marker has more than component (on the one hand X, on the other hand Y), we merely check that all components appear without caring what order they appear in.

Note that this abuses the Python string matching functionality, and assumes that the separator substring never appears in the tokens

class educe.stac.lexicon.pdtb_markers.**Multiword**(*words*)
Bases: object

A sequence of tokens representing a multiword expression.

educe.stac.lexicon.pdtb_markers.**load_pdtb_markers_lexicon**(*filename*)
Load the lexicon of discourse markers from the PDTB.

Parameters **filename** (*str*) – Path to the lexicon.

Returns **markers** – Discourse markers and the relations they signal

Return type dict(*Marker*, list(string))

educe.stac.lexicon.pdtb_markers.**read_lexicon**(*filename*)
Load the lexicon of discourse markers from the PDTB, by relation.

This calls *load_pdtb_markers_lexicon* but inverts the indexing to map each relation to its possible discourse markers.

Note that, as an effect of this inversion, discourse markers whose set of relations is left empty in the lexicon (possibly because they are too ambiguous?) are absent from the inverted index.

Parameters **filename** (*str*) – Path to the lexicon.

Returns **relations** – Relations and their signalling discourse markers.

Return type dict(string, frozenset(*Marker*))

educe.stac.lexicon.wordclass module

Cheap and cheerful lexicon format used in the STAC project. One entry per line, blanks ignored. Each entry associates

- some word with
- some kind of category (we call this a "lexical class")
- an optional part of speech (?? if unknown)
- an optional subcategory blank if none

Here's an example with all four fields

purchase:VBEchange:VB:receivable acquire:VBEchange:VB:receivable give:VBEchange:VB:givable

and one without the notion of subclass

ought:modal:MD: except:negation:??:

class educe.stac.lexicon.wordclass.**LexClass**
Bases: *educe.stac.lexicon.wordclass.LexClass*

Grouping together information for a single lexical class. Our assumption here is that a word belongs to at most one subclass

classmethod **freeze**(*other*)
A frozen copy of a lex class

just_subclasses()
Any subclasses associated with this lexical class

just_words()
Any words associated with this lexical class

classmethod **new_writable_instance**()
A brand new (empty) lex class

class educe.stac.lexicon.wordclass.**LexEntry**
Bases: *educe.stac.lexicon.wordclass.LexEntry*

a single entry in the lexicon

classmethod **read_entries**(*items*)
Return a list of LexEntry given an iterable of entry strings, eg. the stream for the lines in a file. Blank entries are ignored

classmethod **read_entry**(*line*)
Return a LexEntry given the string corresponding to an entry, or raise an exception if we can't parse it

class educe.stac.lexicon.wordclass.**Lexicon**
Bases: *educe.stac.lexicon.wordclass.Lexicon*

All entries in a wordclass lexicon along with some helpers for convenient access

Parameters

- **word_to_subclass** (*Dict String (Dict String String)*) – class to word to subclass nested dict
- **subclasses_to_words** (*Dict String (Set String)*) – class to subclass (to words)

dump()
Print a lexicon's contents to stdout

classmethod read_file(*filename*)
Read the lexical entries in the file of the given name and return a Lexicon

:: FilePath -> IO Lexicon

educe.stac.oneoff package

Toolkit for one-off corpus-editing operations, things we don't expect to come up very frequently, like mass renames of one annotation type to another

Submodules

educe.stac.oneoff.weave module

Combining annotations from an augmented 'source' document (with likely extra text) with those in a 'target' document. This involves copying missing annotations over and shifting the text spans of any matching documents

class educe.stac.oneoff.weave.**Updates**
Bases: *educe.stac.oneoff.weave.Updates*

Expected updates to the target document.

We expect to see four types of annotation:

1. target annotations for which there exists a source annotation in the equivalent span
2. target annotations for which there is no equivalent source annotation (eg. Resources, Preferences, but also annotation moves)
3. source annotations for which there is at least one target annotation at the equivalent span (the mirror to case 1; note that these are not represented in this structure because we don't need to say much about them)
4. source annotations for which there is no match in the target side
5. source annotations that lie in between the matching bits of text

Parameters

- **shift_if_ge** (*dict(int, int)*) – (case 1 and 2) shift points and offsets for characters in the target document (see *shift_spans*)
- **abnormal_src_only** (*[Annotation]*) – (case 4) annotations that only occur in the source document (weird, found in matches)

- **abnormal_tgt_only** ([Annotation]) – (case 2) annotations that only occur in the target document (weird, found in matches)
- **[Annotation]** (*expected_src_only*) – (case 5) annotations that only occur in the source doc (ok, found in gaps)

map(*fun*)
Return an *Updates* in which a function has been applied to all annotations in this one (eg. useful for previewing), and to all spans

exception educe.stac.oneoff.weave.**WeaveException**(**args*, ***kw*)
Bases: exceptions.Exception

Unexpected alignment issues between the source and target document

educe.stac.oneoff.weave.**check_matches**(*tgt_doc*, *matches*, *strict=True*)
Check that the target document text is indeed a subsequence of the source document text (the source document is expected to be "augmented" version of the target with new text interspersed throughout)

Parameters

- **tgt_doc** –
- **matches** (*list of (int, int, int)*) – List of triples (i, j, n) representing matching subsequences: a[i:i+n] == b[j:j+n]. See *difflib.SequenceMatcher.get_matching_blocks*.
- **strict** (*boolean*) – If True, raise an exception if there are match gaps in the target document, otherwise just print the gaps to stderr.

educe.stac.oneoff.weave.**compute_structural_updates**(*src_doc*, *tgt_doc*, *matches*, *updates*, *verbose=0*)
Transfer structural annotations from *tgt_doc* to *src_doc*.

This is the transposition of *compute_updates* to structural units (dialogues only, for the moment).

educe.stac.oneoff.weave.**compute_updates**(*src_doc*, *tgt_doc*, *matches*)
Return updates that would need to be made on the target document.

Given matches between the source and target document, return span updates along with any source annotations that do not have an equivalent in the target document (the latter may indicate that resegmentation has taken place, or that there is some kind of problem)

Parameters

- **src_doc** (Document) –
- **tgt_doc** (Document) –
- **matches** ([Match]) –

Returns updates

Return type *Updates*

educe.stac.oneoff.weave.**find_continuous_seqs**(*doc*, *spans*, *annos*)
Find continuous sequences of annotations, ignoring whitespaces.

Parameters

- **doc** (Document) – Annotated document
- **spans** (*list of Span*) – Spans that support the annotations
- **annos** (*list of Annotation*) – Annotations of interest

- **ignore_whitespaces** (*boolean, optional*) – If True, whitespaces are ignored when assessing continuity.

Returns **seqs** – List of sequences of indices (in annos and spans)

Return type list of list of integers

educe.stac.oneoff.weave.**hollow_out_missing_turn_text**(*src_doc*, *tgt_doc*, *doc_span_src=None*, *doc_span_tgt=None*)

Return a version of the source text where all characters in turns present in *src_doc* but not in *tgt_doc* are replaced with a nonsense char (tab).

Parameters

- **src_doc** (Document) –
- **tgt_doc** (Document) –
- **doc_span_src** (Span, *optional*) –
- **doc_span_tgt** (Span, *optional*) –

Notes

We use difflib's SequenceMatcher to compare the original (but annotated) corpus against the augmented corpus containing nonplayer turns. This gives us the ability to shift annotation spans into the appropriate place within the augmented corpus. By rights the diff should yield only inserts (of the nonplayer turns). But if the inserted text should happen to have the same sorts of substrings as you might find in the rest of corpus, the diff algorithm can be fooled.

educe.stac.oneoff.weave.**shift_char**(*position*, *updates*)

Given a character position an updates tuple, return a shifted over position which reflects the update.

The basic idea that we have a set of "shift points" and their corresponding offsets. If a character position 'c' occurs after one of the points, we take the offset of the largest such point and add it to the character.

Our assumption here is that the update always consists in adding more text so offsets are always positive.

Parameters

- **position** (*int*) – initial position
- **updates** (Updates) –

Returns shifted position

Return type int

educe.stac.oneoff.weave.**shift_dialogues**(*doc_src*, *doc_res*, *updates*, *gen*)

Transpose dialogue split from target to source document.

Remove all dialogues from updates.

Parameters

- **doc_src** (Document) – Source (augmented) document.
- **doc_res** (Document) – Result document, originally a copy of doc_tgt with unshifted annotations. This function modifies *doc_res* by shifting the boundaries of its dialogues according to *updates*, and stretching the first and last dialogues so as to cover the same span as dialogues from *doc_src*.
- **updates** (*set of updates*) – Updates computed by *compute_updates*.

- **gen** (*int*) – Generation of annotations included in *doc_src* and the output.

Returns **updates** – Trimmed down set of *updates*: no more dialogue.

Return type *Updates*

educe.stac.oneoff.weave.**shift_span**(*span*, *updates*, *stretch_right=False*)

Given a span and an updates tuple, return a Span that is shifted over to reflect the updates

Parameters

- **span** (Span) –
- **updates** (Updates) –
- **stretch_right** (*boolean, optional*) – If True, stretch the right boundary of an annotation that buts up against the left of a new annotation. This is recommended for annotations that should fully cover a given span, like dialogues for documents.

Returns **span**

Return type *Span*

See also:

shift_char() for details on how this works

educe.stac.oneoff.weave.**src_gaps**(*matches*)

Given matches between the source and target document, return the spaces between these matches as source offset and size (a bit like the matches). Note that we assume that the target document text is a subsequence of the source document.

educe.stac.oneoff.weave.**stretch_match**(*updates*, *src_doc*, *tgt_doc*, *doc_span_src*, *doc_span_tgt*, *annos_src*, *annos_tgt*, *verbose=0*)

Compute stretch matches between *annos_src* and *annos_tgt*.

Parameters

- **updates** (*Update*) –
- **src_doc** (Document) –
- **tgt_doc** (Document) –
- **doc_span_src** (Span) –
- **doc_span_tgt** (Span) –
- **annos_src** (*list of educe.annotation*) – Unmatched annotations in *span_src*.
- **annos_tgt** (*list of educe.annotation*) – Unmatched annotations in *span_tgt*.
- **verbose** (*int*) – Verbosity level

Returns **res** – Possibly trimmed version of *updates*.

Return type Update

educe.stac.oneoff.weave.**stretch_match_many**(*updates*, *src_doc*, *tgt_doc*, *doc_span_src*, *doc_span_tgt*, *annos_src*, *annos_tgt*, *verbose=0*)

Compute n-m stretch matches between *annos_src* and *annos_tgt*.

Parameters

- **updates** (*Update*) –

- **src_doc** (Document) –
- **tgt_doc** (Document) –
- **doc_span_src** (Span) –
- **doc_span_tgt** (Span) –
- **annos_src** (*list of educe.annotation*) – Unmatched annotations in *span_src*.
- **annos_tgt** (*list of educe.annotation*) – Unmatched annotations in *span_tgt*.
- **verbose** (*int*) – Verbosity level

Returns **res** – Possibly trimmed version of *updates*.

Return type Update

educe.stac.oneoff.weave.**tgt_gaps**(*matches*)
Given matches between the source and target document, return the spaces between these matches as target offset and size (a bit like the matches). By rights this should be empty, but you never know

educe.stac.oneoff.weave.**update_updates**(*updates*, *annos_src*, *annos_tgt*, *verbose=0*)
Update the sets of updates given a match (annos_src, annos_tgt).

Parameters

- **updates** (Updates) – Summary of extra updates between source and target.
- **annos_src** (*list of Annotation*) – Matched annotations from source doc.
- **annos_tgt** (*list of Annotation*) – Matched annotations from target doc.
- **verbose** (*int*) – Verbosity.

Returns **updates** – *updates* updated to take the given match into account.

Return type *Updates*

educe.stac.sanity package

Subpackages

educe.stac.sanity.checks package

Submodules

educe.stac.sanity.checks.annotation module

STAC sanity-check: annotation oversights

class educe.stac.sanity.checks.annotation.**FeatureItem**(*doc*, *contexts*, *anno*, *attrs*, *status='missing'*)
Bases: *educe.stac.sanity.common.ContextItem*

Annotations that are missing some feature(s)

annotations()

html()

educe.stac.sanity.checks.annotation.**is_blank_edu**(*anno*)
True if the annotation looks like it may be an unannotated EDU

educe.stac.sanity.checks.annotation.**is_cross_dialogue**(*contexts*)
The units connected by this relation (or cdu) do not inhabit the same dialogue.

educe.stac.sanity.checks.annotation.**is_fixme**(*feature_value*)
True if a feature value has a fixme value

educe.stac.sanity.checks.annotation.**is_review_edu**(*anno*)
True if the annotation has a FIXME tagged type

educe.stac.sanity.checks.annotation.**missing_features**(*doc*, *anno*)
Return set of attribute names for any expected features that may be missing for this annotation

educe.stac.sanity.checks.annotation.**run**(*inputs*, *k*)
Add any annotation omission errors to the current report

educe.stac.sanity.checks.annotation.**search_for_fixme_features**(*inputs*, *k*)
Return a ReportItem for any annotations in the document whose features have a fixme type

educe.stac.sanity.checks.annotation.**search_for_missing_rel_feats**(*inputs*, *k*)
Return ReportItems for any relations that are missing expected features

educe.stac.sanity.checks.annotation.**search_for_missing_unit_feats**(*inputs*, *k*)
Return ReportItems for any EDUs and CDUs that are missing expected features

educe.stac.sanity.checks.annotation.**search_for_unexpected_feats**(*inputs*, *k*)
Return ReportItems for any annotations that are have features we were not expecting them to have

educe.stac.sanity.checks.annotation.**unexpected_features**(*_*, *anno*)
Return set of attribute names for any features that we were not expecting to see in the given annotations

educe.stac.sanity.checks.glozz module

Sanity checker: low-level Glozz errors

class educe.stac.sanity.checks.glozz.**BadIdItem**(*doc*, *contexts*, *anno*, *expected_id*)
Bases: *educe.stac.sanity.common.ContextItem*

An annotation whose identifier does not match its metadata

text()

class educe.stac.sanity.checks.glozz.**DuplicateItem**(*doc*, *contexts*, *anno*, *others*)
Bases: *educe.stac.sanity.common.ContextItem*

An annotation which shares an id with another

text()

class educe.stac.sanity.checks.glozz.**IdMismatch**(*doc*, *contexts*, *unit1*, *unit2*)
Bases: *educe.stac.sanity.common.ContextItem*

An annotation which seems to have an equivalent in some twin but with the wrong identifier

annotations()

html()

exception educe.stac.sanity.checks.glozz.**MissingDocumentException**(*k*)
Bases: exceptions.Exception

A document we are trying to cross check does not have the expected twin

class educe.stac.sanity.checks.glozz.**MissingItem**(*status*, *doc1*, *contexts1*, *unit*, *doc2*, *contexts2*, *approx*)

Bases: *educe.stac.sanity.report.ReportItem*

An annotation which is missing in some document twin (or which looks like it may have been unexpectedly added)

excess_status = 'ADDED'

html()

missing_status = 'DELETED'

status_len = 7

text_span()

Return the span for the annotation in question

class educe.stac.sanity.checks.glozz.**OffByOneItem**(*doc*, *contexts*, *unit*)

Bases: *educe.stac.sanity.common.UnitItem*

An annotation whose boundaries might be off by one

html()

html_turn_info(*parent*, *turn*)

Given a turn annotation, append a prettified HTML representation of the turn text (highlighting parts of it, such as the turn number)

class educe.stac.sanity.checks.glozz.**OverlapItem**(*doc*, *contexts*, *anno*, *overlaps*)

Bases: *educe.stac.sanity.common.ContextItem*

An annotation whose span overlaps with that of another

annotations()

html()

educe.stac.sanity.checks.glozz.**bad_ids**(*inputs*, *k*)

Return annotations whose identifiers do not match their metadata

educe.stac.sanity.checks.glozz.**check_unit_ids**(*inputs*, *key1*, *key2*)

Return annotations that match in the two documents modulo identifiers. This might arise if somebody creates a duplicate annotation in place and annotates that

educe.stac.sanity.checks.glozz.**cross_check_against**(*inputs*, *key1*, *stage='unannotated'*)

Compare annotations with their equivalents on a twin document in the corpus

educe.stac.sanity.checks.glozz.**cross_check_units**(*inputs*, *key1*, *key2*, *status*)

Return tuples for certain corpus[key1] units not present in corpus[key2]

educe.stac.sanity.checks.glozz.**duplicate_annotations**(*inputs*, *k*)

Multiple annotations with the same local_id()

educe.stac.sanity.checks.glozz.**filter_matches**(*unit*, *other_units*)

Return any unit-level annotations in *other_units* that look like they may be the same as the given annotation

educe.stac.sanity.checks.glozz.**is_maybe_off_by_one**(*text*, *anno*)

True if an annotation has non-whitespace characters on its immediate left/right

educe.stac.sanity.checks.glozz.**overlapping**(*inputs*, *k*, *is_overlap*)

Return items for annotations that have overlaps

educe.stac.sanity.checks.glozz.**overlapping_structs**(*inputs*, *k*)

Return items for structural annotations that have overlaps

educe.stac.sanity.checks.glozz.**run**(*inputs*, *k*)
Add any glozz errors to the current report

educe.stac.sanity.checks.glozz.**search_glozz_off_by_one**(*inputs*, *k*)
EDUs which have non-whitespace (or boundary) characters either on their right or left

educe.stac.sanity.checks.graph module

Sanity checker: fancy graph-based errors

educe.stac.sanity.checks.graph.**BACKWARDS_WHITELIST = ['Conditional']**
relations that are allowed to go backwards

class educe.stac.sanity.checks.graph.**CduOverlapItem**(*doc*, *contexts*, *anno*, *cdus*)
Bases: *educe.stac.sanity.common.ContextItem*

EDUs that appear in more than one CDU

annotations()

html()

educe.stac.sanity.checks.graph.**PAIRS_WHITELIST = [('Contrast', 'Comment'), ('Narration', 'Result'), ('Narratio**
pairs of relations that are explicitly allowed between the same source/target DUs

educe.stac.sanity.checks.graph.**are_single_headed_cdus**(*inputs*, *k*, *gra*)
Check that each CDU has exactly one head DU.

Parameters **gra** (Graph) – Graph for the discourse structure.

Returns **report_items** – List of report items, one per faulty CDU.

Return type list of ReportItem

educe.stac.sanity.checks.graph.**dialogue_graphs**(*k*, *doc*, *contexts*)
Return a dict from dialogue annotations to subgraphs containing at least everything in that dialogue (and perhaps some connected items).

Parameters

- **k** (FileId) – File identifier
- **doc** (*TODO*) – TODO
- **contexts** (*dict(Annotation, Context)*) – Context for each annotation.

Returns **graphs** – Graph for each dialogue.

Return type dict(*Dialogue*, *Graph*)

Notes

MM: I could not find any caller for this function in either educe or irit-stac, as of 2017-03-17.

educe.stac.sanity.checks.graph.**horrible_context_kludge**(*graph*, *simplified_graph*, *contexts*)
Given a graph and its copy, and given a context dictionary, return a copy of the context dictionary that corresponds to the simplified graph. Ugh

educe.stac.sanity.checks.graph.**is_arrow_inversion**(*gra*, *_*, *rel*)
Relation in a graph that goes from textual right to left (may not be a problem)

educe.stac.sanity.checks.graph.**is_bad_relset**(*gra*, *contexts*, *relset*)
True if a set of relation instances has more than one member and it is not whitelisted.

Parameters

- **gra** (Graph) – Graph for the discourse structure.
- **contexts** (*TODO*) – TODO
- **relset** (*set of relation instances*) – Set of relation instances on the same DUs ; each instance is a pair (udir, rel), where: udir is one of {'src_tgt', 'tgt_src'} and rel is the identifier of a relation.

Returns **res** – True if relset contains more than one element and *is_whitelisted_relpair* returns False.

Return type boolean

educe.stac.sanity.checks.graph.**is_disconnected**(*gra*, *contexts*, *node*)
Return True if an EDU is disconnected from a discourse structure.

An EDU is considered disconnected unless:

- it has an incoming link or
- it has an outgoing Conditional link or
- it's at the beginning of a dialogue

In principle we don't need to look at EDUs that are disconnected on the outgoing end because (1) it can be legitimate for non-dialogue-ending EDUs to not have outgoing links and (2) such information would be redundant with the incoming anyway.

educe.stac.sanity.checks.graph.**is_dupe_rel**(*gra*, *_*, *rel*)
Relation instance for which there are relation instances between the same source/target DUs (regardless of direction)

educe.stac.sanity.checks.graph.**is_non2sided_rel**(*gra*, *_*, *rel*)
Relation instance which does not have exactly a source and target link in the graph

How this can possibly happen is a mystery

educe.stac.sanity.checks.graph.**is_puncture**(*gra*, *_*, *rel*)
Relation in a graph that traverse a CDU boundary

educe.stac.sanity.checks.graph.**is_weird_ack**(*gra*, *contexts*, *rel*)
Relation in a graph that represent a question answer pair which either does not start with a question, or which ends in a question.

Note the detection process is a lot sloppier when one of the endpoints is a CDU. If all EDUs in the CDU are by the same speaker, we can check as usual; otherwise, all bets are off, so we ignore the relation.

Note: slightly curried to accept contexts as an argument

educe.stac.sanity.checks.graph.**is_weird_qap**(*gra*, *contexts*, *rel*)
Return True if rel is a weird Question-Answer Pair relation.

Parameters

- **gra** (*TODO*) – Graph?
- **contexts** (*TODO*) – Surrounding context
- **rel** (*TODO*) – Relation.

Returns **res** – True if rel is a relation that represents a question answer pair which either does not start with a question, or which ends in a question.

Return type boolean

educe.stac.sanity.checks.graph.**is_whitelisted_relpair**(*gra*, *_*, *relset*)
True if a pair of instance relations is in *PAIRS_WHITELIST*.

Parameters

- **gra** (Graph) – Graph for the discourse structure.
- **contexts** (*TODO*) – TODO
- **relset** (*set of relation instances*) – Set of relation instances on the same DUs ; each instance is a pair (udir, rel), where: udir is one of {'src_tgt', 'tgt_src'} and rel is the identifier of a relation.

Returns **res** – True if relset is a pair of relation instances with the same direction and the corresponding pair of relations is explicitly allowed in the whitelist.

Return type boolean

educe.stac.sanity.checks.graph.**rel_link_item**(*doc*, *contexts*, *gra*, *rel*)
return ReportItem for a graph relation

educe.stac.sanity.checks.graph.**rfc_violations**(*inputs*, *k*, *gra*)
Repackage right frontier contraint violations in a somewhat friendlier way

educe.stac.sanity.checks.graph.**run**(*inputs*, *k*)
Add any graph errors to the current report

educe.stac.sanity.checks.graph.**search_graph_cdu_overlap**(*inputs*, *k*, *gra*)
Return a ReportItem for every EDU that appears in more than one CDU

educe.stac.sanity.checks.graph.**search_graph_cdus**(*inputs*, *k*, *gra*, *pred*)
Return a ReportItem for any CDU in the graph for which the given predicate is True

educe.stac.sanity.checks.graph.**search_graph_edus**(*inputs*, *k*, *gra*, *pred*)
Return a ReportItem for any EDU within the graph for which some predicate is true

educe.stac.sanity.checks.graph.**search_graph_relations**(*inputs*, *k*, *gra*, *pred*)
Return a ReportItem for any relation instance within the graph for which some predicate is true

educe.stac.sanity.checks.graph.**search_graph_relations_same_dus**(*inputs*, *k*, *gra*, *pred*)
Return a list of ReportItem (one per member of the set) for any set of relation instances within the graph for which some predicate is True.

Parameters

- **inputs** (educe.stac.sanity.main.SanityChecker) – SanityChecker, with attributes *corpus* and *contexts*.
- **k** (FileId) – Identifier of the desired Glozz document.
- **gra** (educe.stac.graph.Graph) – Graph that corresponds to the discourse structure (?).
- **pred** (*function from (gra, contexts, rel_set) to boolean*) – Predicate function.

Returns **report_items** – One ReportItem for each relation instance that belongs to a set of instances, on the same DUs, where pred is True.

Return type list of ReportItem

educe.stac.sanity.checks.type_err module

STAC sanity-check: type errors

educe.stac.sanity.checks.type_err.**has_non_du_member**(*anno*)
True if *anno* is a relation that points to another relation, or if it's a CDU that has relation members

educe.stac.sanity.checks.type_err.**is_non_du**(*anno*)
True if the annotation is neither an EDU nor a CDU

educe.stac.sanity.checks.type_err.**is_non_preference**(*anno*)
True if the annotation is NOT a preference

educe.stac.sanity.checks.type_err.**is_non_resource**(*anno*)
True if the annotation is NOT a resource

educe.stac.sanity.checks.type_err.**run**(*inputs*, *k*)
Add any annotation type errors to the current report

educe.stac.sanity.checks.type_err.**search_anaphora**(*inputs*, *k*, *pred*)
Return a ReportItem for any anaphora annotation in which at least one member (not the annotation itself) is true with the given predicate

educe.stac.sanity.checks.type_err.**search_preferences**(*inputs*, *k*, *pred*)
Return a ReportItem for any Preferences schema which has at least one member for which the predicate is True

educe.stac.sanity.checks.type_err.**search_resource_groups**(*inputs*, *k*, *pred*)
Return a ReportItem for any Several_resources schema which has at least one member for which the predicate is True

Submodules

educe.stac.sanity.common module

Functionality and report types common to sanity checker

class educe.stac.sanity.common.**ContextItem**(*doc*, *contexts*)
Bases: *educe.stac.sanity.report.ReportItem*

Report item involving EDU contexts

class educe.stac.sanity.common.**RelationItem**(*doc*, *contexts*, *rel*, *naughty*)
Bases: *educe.stac.sanity.common.ContextItem*

Errors which involve Glozz relation annotations

annotations()

html()

class educe.stac.sanity.common.**SchemaItem**(*doc*, *contexts*, *schema*, *naughty*)
Bases: *educe.stac.sanity.common.ContextItem*

Errors which involve Glozz schema annotations

annotations()

html()

class educe.stac.sanity.common.**UnitItem**(*doc*, *contexts*, *unit*)
Bases: *educe.stac.sanity.common.ContextItem*

Errors which involve Glozz unit-level annotations

annotations()

html()

educe.stac.sanity.common.**anno_code**(*anno*)
Short code providing a clue what the annotation is

educe.stac.sanity.common.**is_default**(*anno*)
True if the annotation has type 'default'

educe.stac.sanity.common.**is_glozz_relation**(*anno*)
True if the annotation is a Glozz relation

educe.stac.sanity.common.**is_glozz_schema**(*anno*)
True if the annotation is a Glozz schema

educe.stac.sanity.common.**is_glozz_unit**(*anno*)
True if the annotation is a Glozz unit

educe.stac.sanity.common.**rough_type**(*anno*)
Return either

- "EDU"
- "relation"
- or the annotation type

educe.stac.sanity.common.**search_for_glozz_relations**(*inputs*, *k*, *pred*, *endpoint_is_naughty=None*)
Return a `ReportItem` for any glozz relation that satisfies the given predicate.

If *endpoint_is_naughty* is supplied, note which of the endpoints can be considered naughty

educe.stac.sanity.common.**search_for_glozz_schema**(*inputs*, *k*, *pred*, *member_is_naughty=None*)
Search for schema that satisfy a condition

educe.stac.sanity.common.**search_glozz_units**(*inputs*, *k*, *pred*)
Return an item for every unit-level annotation in the given document that satisfies some predicate

Return type `ReportItem`

educe.stac.sanity.common.**search_in_glozz_schema**(*inputs*, *k*, *stype*, *pred*, *member_is_naughty=None*)
Search for schema whose memmbers satisfy a condition. Not to be confused with *search_for_glozz_schema*

educe.stac.sanity.common.**summarise_anno**(*doc*, *light=False*)
Return a function that returns a short text summary of an annotation

educe.stac.sanity.common.**summarise_anno_html**(*doc*, *contexts*)
Return a function that creates HTML descriptions of an annotation given document and contexts

educe.stac.sanity.html module

Helpers for building HTML Hint: import the ET for the ET package too

educe.stac.sanity.html.**br**(*parent*)
Create and return an HTML br tag under the parent node

educe.stac.sanity.html.**elem**(*parent*, *tag*, *text=None*, *attrib=None*, ***kwargs*)
Create an HTML element under the given parent node, with some text inside of it

educe.stac.sanity.html.**span**(*parent*, *text=None*, *attrib=None*, ***kwargs*)
Create and return an HTML span under the given parent node

educe.stac.sanity.main module

Check the corpus for any consistency problems

class educe.stac.sanity.main.**SanityChecker**(*args*)
Bases: object

Sanity checker settings and state

output_is_temp()
True if we are writing to an output directory

run()
Perform sanity checks and write the output

educe.stac.sanity.main.**add_element**(*settings*, *k*, *html*, *descr*, *mk_path*)
Add a link to a report element for a given document, but only if it actually exists

educe.stac.sanity.main.**copy_parses**(*settings*)
Copy relevant stanford parser outputs from corpus to report

educe.stac.sanity.main.**create_dirname**(*path*)
Create the directory beneath a path if it does not exist

educe.stac.sanity.main.**easy_settings**(*args*)
Modify args to reflect user-friendly defaults.

Terminates the program if *args.corpus* is set but does not point to an existing folder ; otherwise *args.doc* must be set and everything else is expected to be empty.

Parameters **args** (*Namespace*) – Arguments of the argparser.

See also:

educe.stac.util.args.check_easy_settings()

educe.stac.sanity.main.**first_or_none**(*itrs*)
Return the first element or None if there isn't one

educe.stac.sanity.main.**generate_graphs**(*settings*)
Draw SVG graphs for each of the documents in the corpus

educe.stac.sanity.main.**issues_descr**(*report*, *k*)
Return a string characterising a report as either being warnings or error (helps the user scan the index to figure out what needs clicking on)

educe.stac.sanity.main.**main**()
Sanity checker CLI entry point

educe.stac.sanity.main.**run_checks**(*inputs*, *k*)
Run sanity checks for a given document

educe.stac.sanity.main.**sanity_check_order**(*k*)
We want to sort file id by order of

1. doc
2. subdoc
3. annotator

4. stage (unannotated < unit < discourse)

The important bit here is the idea that we should maybe group unit and discourse for 1-3 together

educe.stac.sanity.main.**write_index**(*settings*)
Write the report index

educe.stac.sanity.report module

Reporting component of sanity checker

class educe.stac.sanity.report.**HtmlReport**(*anno_files*, *output_dir*)
Bases: object

Representation of a report that we would like to generate. Output will be dumped to a directory

anchor_name(*k*, *header*)
HTML anchor name for a report section

css = '\n.annoid { font-family: monospace; font-size: small; }\n.feature { font-family: monospace; }\n.snippet { font-style:

delete(*k*)
Delete the subreport for a given key. This can be used if you want to iterate through lots of different keys, generating reports incrementally and then deleting them to avoid building up memory.

No-op if we don't have a sub-report for the given key

flush_subreport(*k*)
Write and delete (to save memory)

has_errors(*k*)
If we have error-level reports for the given key

javascript = '\nfunction has(xs, x) {\n for (e in xs) {\n if (xs[e] === x) { return true; }\n }\n return false;\n}\n\n\nfunctio

mk_hidden_with_toggle(*parent*, *anchor*)
Attach some javascript and html to the given block-level element that turns it into a hide/show toggle block, starting out in the hidden state

mk_or_get_subreport(*k*)
Initialise and cache the subreport for a key, including the subreports for each severity level below it

If already cached, retrieve from cache

classmethod mk_output_path(*odir*, *k*, *extension=''*)
Generate a path within a parent directory, given a fileid

report(*k*, *err_type*, *severity*, *header*, *items*, *noisy=False*)
Append bullet points for each item to the appropriate section of the appropriate report in progress

set_has_errors(*k*)
Note that this report has seen at least one error-level severity message

subreport_path(*k*, *extension='.report.html'*)
Report for a single document

write(*k*, *path*)
Write the subreport for a given key to the path. No-op if we don't have a sub-report for the given key

class educe.stac.sanity.report.**ReportItem**
Bases: object

An individual reportable entry (usually involves a list of annotations), rendered as a block of text in the report

annotations()
The annotations which this report item is about

html()
Return an HTML element corresponding to the visualisation for this item

text()
If you don't want to create an HTML visualisation for a report item, you can fall back to just generating lines of text

Return type [string]

class educe.stac.sanity.report.**Severity**
Bases: enum.Enum

Severity of a sanity check error block

error = 2

warning = 1

class educe.stac.sanity.report.**SimpleReportItem**(*lines*)
Bases: *educe.stac.sanity.report.ReportItem*

Report item which just consists of lines of text

text()

educe.stac.sanity.report.**html_anno_id**(*parent*, *anno*, *bracket=False*)
Create and return an HTML span parent node displaying the local annotation id for an annotation item

educe.stac.sanity.report.**mk_microphone**(*report*, *k*, *err_type*, *severity*)
Return a convenience function that generates report entries at a fixed error type and severity level

Return type (string, [*ReportItem*]) -> string

educe.stac.sanity.report.**snippet**(*txt*, *stop=50*)
truncate a string if it's longer than *stop* chars

educe.stac.util package

Submodules

educe.stac.util.annotate module

Readable text dumps of educe annotations.

The idea here is to dump the text to screen, and use some informal text markup to show annotations over the text. There's a limit to how much we can display, but just breaking things up into paragraphs and [segments] seems to go a long way.

educe.stac.util.annotate.**annotate**(*txt*, *annotations*, *inserts=None*)
Decorate a text with arbitrary bracket symbols, as a visual guide to the annotations on that text. For example, in a chat corpus, you might use newlines to indicate turn boundaries and square brackets for segments.

Parameters

- **inserts** – inserts a dictionary from annotation type to pair of its opening/closing bracket
- **FIXME**(*this needs to become a standard educe utility,*) –
- **as part of the educe.annotation layer?**(*maybe*) –

educe.stac.util.annotate.**annotate_doc**(*doc*, *span=None*)
Pretty print an educe document and its annotations.

See the lower-level *annotate* for more details

educe.stac.util.annotate.**reflow**(*text*, *width=40*)
Wrap some text, at the same time ensuring that all original linebreaks are still in place

educe.stac.util.annotate.**rough_type**(*anno*)
Simplify STAC annotation types

educe.stac.util.annotate.**schema_text**(*doc*, *anno*)
(recursive) text preview of a schema and its contents. Members are enclosed in square brackets.

educe.stac.util.annotate.**show_diff**(*doc_before*, *doc_after*, *span=None*)
Display two educe documents (presumably two versions of the "same") side by side

educe.stac.util.args module

Command line options

educe.stac.util.args.**add_commit_args**(*parser*)
Augment a subcommand argparser with an option to emit a commit message for your version control tracking

educe.stac.util.args.**add_usual_input_args**(*parser*, *doc_subdoc_required=False*, *help_suffix=None*)
Augment a subcommand argparser with typical input arguments. Sometimes your subcommand may require slightly different input arguments, in which case, just don't call this function.

Parameters

- **parser** (*ArgumentParser*) – Argument parser.
- **doc_subdoc_required** (*bool, defaults to False*) – force user to supply –doc/–subdoc for this subcommand (note you'll need to add stage/anno yourself)
- **help_suffix** (*string, defaults to None*) – appended to –doc/–subdoc help strings

educe.stac.util.args.**add_usual_output_args**(*parser*, *default_overwrite=False*)
Augment a subcommand argparser with typical output arguments, Sometimes your subcommand may require slightly different output arguments, in which case, just don't call this function.

educe.stac.util.args.**anno_id**(*string*)
Split AUTHOR_DATE string into tuple, complaining if we don't have such a string. Used for argparse

educe.stac.util.args.**announce_output_dir**(*output_dir*)
Tell the user where we saved the output

educe.stac.util.args.**check_easy_settings**(*args*)
Modify args to reflect user-friendly defaults.

Terminates the program if *args.corpus* is set but does not point to an existing folder ; otherwise *args.doc* must be set and everything else is expected to be empty.

Notes

All callers for this function are in the *scripts* folder of the educe repository: scripts/stac-{util,edit,oneoff}.

Parameters **args** (*Namespace*) – Arguments of the argparser.

See also:

educe.stac.sanity.main.easy_settings()

`educe.stac.util.args.`**`comma_span`**(*string*)
Split a comma delimited pair of integers into an educe span

`educe.stac.util.args.`**`get_output_dir`**(*args*, *default_overwrite=False*)
Return the output dir specified or inferred from command line args.

We try the following in order:

1. If *–output* is given explicitly, we'll just use/create that
2. If *default_overwrite* is True, or the user specifies *–overwrite* on the command line (provided the command supports it), the output directory may well be the original corpus dir (*gulp*! Better use version control!)
3. OK just make a temporary directory. Later on, you'll probably want to call *announce_output_dir*.

`educe.stac.util.args.`**`read_corpus`**(*args*, *preselected=None*, *verbose=True*)
Read the section of the corpus specified in the command line arguments.

`educe.stac.util.args.`**`read_corpus_with_unannotated`**(*args*, *verbose=True*)
Read the section of the corpus specified in the command line arguments.

educe.stac.util.csv module

educe.stac.util.doc module

Utilities for large-scale changes to educe documents, for example, moving a chunk of text from one document to another

exception `educe.stac.util.doc.`**`StacDocException`**(*msg*)
Bases: `exceptions.Exception`

An exception that arises from trying to manipulate a stac document (typically moving things around, etc)

`educe.stac.util.doc.`**`compute_renames`**(*avoid*, *incoming*)
Given two sets of documents (i.e. corpora), return a dictionary which would allow us to rename ids in *incoming* so that they do not overlap with those in *avoid*.

:rtype *author -> date -> date*

`educe.stac.util.doc.`**`evil_set_id`**(*anno*, *author*, *date*)
This is a bit evil as it's using undocumented functionality from the educe.annotation.Standoff object

`educe.stac.util.doc.`**`evil_set_text`**(*doc*, *text*)
This is a bit evil as it's using undocumented functionality from the educe.annotation.Document object

`educe.stac.util.doc.`**`move_portion`**(*renames*, *src_doc*, *tgt_doc*, *src_split*, *tgt_split=-1*)
Move part of the source document into the target document.

This returns an updated copy of both the source and target documents.

This can capture a couple of patterns:

- reshuffling the boundary between the target and source document (if *tgt | src1 src2 ==> tgt src1 | src2*) (*tgt_split = -1*)
- prepending the source document to the target (*src | tgt ==> src tgt*; *src_split=-1; tgt_split=0*)
- inserting the whole source document into the other (*tgt1 tgt2 + src ==> tgt1 src tgt2*; *src_split=-1*)

There's a bit of potential trickiness here:

- we'd like to preserve the property that text has a single starting and ending space (no real reason just seems safer that way)
- if we're splicing documents together particularly at their respective ends, there's a strong off-by-one risk because some annotations span the whole text (whitespace and all), particularly dialogues

Parameters

- **renames** (*TODO*) – TODO
- **src_doc** (Document) – Source document
- **tgt_doc** (Document) – Target document
- **src_split** (*int*) – Split point for *src_doc*.
- **tgt_split** (*int, defaults to -1*) – Split point for *tgt_doc*.

Returns

- **new_src_doc** (*Document*) – TODO
- **new_tgt_doc** (*Document*) – TODO

educe.stac.util.doc.**narrow_to_span**(*doc*, *span*)
Return a deep copy of a document with only the text and annotations that are within the span specified by portion.

educe.stac.util.doc.**rename_ids**(*renames*, *doc*)
Return a deep copy of a document, with ids reassigned according to the renames dictionary

educe.stac.util.doc.**retarget**(*doc*, *old_id*, *new_anno*)
Replace all links to the old (unit-level) annotation with links to the new one.

We refer to the old annotation by id, but the new annotation must be passed in as an object. It must also be either an EDU or a CDU.

Return True if we replaced anything

educe.stac.util.doc.**shift_annotations**(*doc*, *offset*, *point=None*)
Return a deep copy of a document such that all annotations have been shifted by an offset.

If shifting right, we pad the document with whitespace to act as filler. If shifting left, we cut the text.

If a shift point is specified and the offset is positive, we only shift annotations that are to the right of the point. Likewise if the offset is negative, we only shift those that are to the left of the point.

educe.stac.util.doc.**split_doc**(*doc*, *middle*)
Given a split point, break a document into two pieces.

If the split point is None, we take the whole document (this is slightly different from having -1 as a split point)

Raise an exception if there are any annotations that span the point.

Parameters

- **doc** (Document) – The document we want to split.
- **middle** (*int*) – Split point.

Returns

- **doc_prefix** (*Document*) – Deep copy of *doc* restricted to span [:middle]
- **doc_suffix** (*Document*) – Deep copy of *doc* restricted to span [middle:] ; the span of each annotation is shifted to match the new text.

educe.stac.util.doc.**strip_fixme**(*act*)
Remove the fixme string from a dialogue act annotation. These were automatically inserted when there is an annotation to review. We shouldn't see them for any use cases like feature extraction though.

See *educe.stac.dialogue_act* which returns the set of dialogue acts for each annotation (by rights should be singleton set, but there used to be more than one, something we want to phase out?)

educe.stac.util.doc.**unannotated_key**(*key*)
Given a corpus key, return a copy of that equivalent key in the unannotated portion of the corpus (the parser outputs objects that are based in unannotated)

educe.stac.util.glozz module

STAC Glozz conventions

class educe.stac.util.glozz.**PseudoTimestamper**
Bases: object

Generator for the fake timestamps used as a Glozz IDs

next()
Fresh timestamp

class educe.stac.util.glozz.**TimestampCache**
Bases: object

Generates and stores a unique timestamp entry for each key. You can use any hashable key, for exmaple, a span, or a turn id.

get(*tid*)
Return a timestamp for this turn id, either generating and caching (if unseen) or fetching from the cache

reset()
Empty the cache (but maintain the timestamper state, so that different documents get different timestamps; the difference in timestamps is not mission-critical but potentially nice)

educe.stac.util.glozz.**anno_author**(*anno*)
Annotation author

educe.stac.util.glozz.**anno_date**(*anno*)
Annotation creation date as an int

educe.stac.util.glozz.**anno_id_from_tuple**(*author_date*)
Glozz string representation of authors and dates (AUTHOR_DATE)

educe.stac.util.glozz.**anno_id_to_tuple**(*string*)
Read a Glozz string representation of authors and dates into a pair (date represented as an int, ms since 1970?)

educe.stac.util.glozz.**get_turn**(*tid*, *doc*)
Return the turn annotation with the desired ID

educe.stac.util.glozz.**is_dialogue**(*anno*)
If a Glozz annotation is a STAC dialogue.

educe.stac.util.glozz.**set_anno_author**(*anno*, *author*)
Replace the annotation author the given author

educe.stac.util.glozz.**set_anno_date**(*anno*, *date*)
Replace the annotation creation date with the given integer

educe.stac.util.output module

Help writing out corpus files

`educe.stac.util.output.`**`mk_parent_dirs`**(*filename*)
Given a filepath that we want to write, create its parent directory as needed.

`educe.stac.util.output.`**`output_path_stub`**(*odir*, *k*)
Given an output directory and an educe corpus key, return a 'stub' output path in that directory. This is dirname and basename only; you probably want to tack a suffix onto it.

Example: given something like "/tmp/foo" and a key like *{author:"bob", stage:units, doc:"pilot03", subdoc:"07"}* you might get something like */tmp/foo/pilot03/units/pilot03_07*)

`educe.stac.util.output.`**`save_document`**(*output_dir*, *k*, *doc*)
Save a document as a Glozz .ac/.aa pair

`educe.stac.util.output.`**`write_dot_graph`**(*doc_key*, *odir*, *dot_graph*, *part=None*, *run_graphviz=True*)
Write a dot graph and possibly run graphviz on it

educe.stac.util.prettifyxml module

Function to "prettify" XML: courtesy of http://www.doughellmann.com/PyMOTW/xml/etree/ElementTree/create.html

`educe.stac.util.prettifyxml.`**`prettify`**(*elem*, *indent=''*)
Return a pretty-printed XML string for the Element.

educe.stac.util.showscores module

class `educe.stac.util.showscores.`**`Score`**(*reference*, *test*)
Precision/recall type scores for a given data set.

This class is really just about holding on to sets of things. The actual maths is handled by NLTK.

`f_measure`()

`missing`()

`precision`()

`recall`()

`shared`()

`spurious`()

`educe.stac.util.showscores.`**`banner`**(*t*)

`educe.stac.util.showscores.`**`show_multi`**(*k*, *score*)

`educe.stac.util.showscores.`**`show_pair`**(*k*, *score*)

Submodules

educe.stac.annotation module

STAC annotation conventions (re-exported in educe.stac)

STAC/Glozz annotations can be a bit confusing because for two reasons, first that Glozz objects are used to annotate very different things; and second that annotations are done on different stages

Stage 1 (units)

Glozz	Uses
units	doc structure, EDUs, resources, preferences
relations	coreference
schemas	composite resources

Stage 2 (discourse)

Glozz	Uses
units	doc structure, EDUs
relations	relation instances, coreference
schemas	CDUs

Units

There is a typology of unit types worth noting:

- doc structure : type eg. *Dialogue*, *Turn*, *paragraph*
- resources : subspans of segments (type *Resource*)
- preferences : subspans of segments (type *Preference*)
- EDUs : spans of text associated with a dialogue act (eg. type *Offer*, *Accept*) (during discourse stage, these are just type *Segment*)

Relations

- coreference : (type *Anaphora*)
- relation instances : links between EDUs, annotated with relation label (eg. type *Elaboration*, type *Contrast*, etc). These can be further divided in subordinating or coordination relation instances according to their label

Schemas

- composite resources : boolean combinations of resources (eg. "sheep or ore")
- CDUs: type *Complex_discourse_unit* (discourse stage)

class `educe.stac.annotation.`**`PartialUnit`**
Bases: *educe.stac.annotation.PartialUnit*

Partially instantiated unit, for use when you want to programmatically insert annotations into a document

A partially instantiated unit does not have any metadata (creation date, etc); as these will be derived automatically

`educe.stac.annotation.`**`RENAMES = {'Strategic_comment': 'Other', 'Segment': 'Other'}`**
Dialogue acts that should be treated as a different one

class `educe.stac.annotation.`**`TurnId`**
Bases: `tuple`

Turn identifier akin to a Gorn address.

A Gorn address is a tuple of integers.

classmethod `from_string`(*tid_str*)
Create a TurnId from a string.

ex: (21.0.1)

educe.stac.annotation.**addressees**(*anno*)
The set of people spoken to during an edu annotation

```
Annotation -> Set String
```

Note: this returns *None* if the value is the default 'Please choose...'; but otherwise, it preserves values like 'All' or '?'.

educe.stac.annotation.**cleanup_comments**(*anno*)
Strip out default comment text from features. This placeholder text was inserted as a UI aid during editing in Glozz, but isn't actually the comment itself

educe.stac.annotation.**create_units**(*_*, *doc*, *author*, *partial_units*)
Create a collection of units from their partial specification.

Parameters

- **_** (*anything*) – Anonymous parameter whose value is ignored. It was apparently supposed to contain a FileId. I suppose the intention was to follow a signature similar to other functions.
- **doc** (Document) – Containing document.
- **author** (*string*) – Author for the new units.
- **partial_units** (iterable of *PartialUnit*) – Partial specification of the new units.

Returns **res** – Collection of instantiated new unit objects.

Return type list of Unit

Notes

As of 2016-05-11, this function does not seem to be used anymore in the codebase. It used to be called in *irit-stac/segmentation/glozz-segment*, which was deleted 2015-06-08 (commit e2373c03) because it was not used.

educe.stac.annotation.**dialogue_act**(*anno*)
Set of dialogue act (aka speech act) annotations for a Unit, taking into consideration STAC conventions like collapsing Strategic_comment into Other

By rights should be singleton set, but there used to be more than one, something we want to phase out?

educe.stac.annotation.**game_turns**(*doc*, *turns*, *gen=2*)
Group a sequence of turns into a sequence of game turns.

A game turn corresponds to the sequence of events (turns) that happen within a player's turn (in the SOC game).

Parameters

- **doc** (Document) – Containing document.
- **turns** (*list of educe.stac.Unit*) – Events (of type Turn) from the game: server messages, player messages.

Returns **gturn_beg** – Index of the first Turn of each game turn.

Return type list of int

educe.stac.annotation.**is_cdu**(*annotation*)
See CDUs typology above

educe.stac.annotation.**is_coordinating**(*annotation*)
See Relation typology above

educe.stac.annotation.**is_dialogue**(*annotation*)
See Unit typology above

educe.stac.annotation.**is_dialogue_act**(*annotation*)
Deprecated in favour of is_edu

educe.stac.annotation.**is_edu**(*annotation*)
See Unit typology above

educe.stac.annotation.**is_paragraph**(*annotation*)
See Unit typology above

educe.stac.annotation.**is_preference**(*annotation*)
See Unit typology above

educe.stac.annotation.**is_relation_instance**(*annotation*)
See Relation typology above

educe.stac.annotation.**is_resource**(*annotation*)
See Unit typology above

educe.stac.annotation.**is_structure**(*annotation*)
Is one of the document-structure annotations, something an annotator is expected not to edit, create, delete

educe.stac.annotation.**is_subordinating**(*annotation*)
See Relation typology above

educe.stac.annotation.**is_turn**(*annotation*)
See Unit typology above

educe.stac.annotation.**is_turn_star**(*annotation*)
See Unit typology above

educe.stac.annotation.**relation_labels**(*anno*)
Set of relation labels (eg. Elaboration, Explanation), taking into consideration any applicable STAC-isms

educe.stac.annotation.**set_addressees**(*anno*, *addr*)
Set the addresee list for an annotation. If the value *None* is provided, the addressee list is deleted (if present)

```
(Iterable String, Annotation) -> IO ()
```

educe.stac.annotation.**speaker**(*anno*)
Return the speaker associated with a turn annotation. NB: crashes if there is none

educe.stac.annotation.**split_turn_text**(*text*)
STAC turn texts are prefixed with a turn number and speaker to help the annotators (eg. "379: Bob: I think it's your go, Alice").

Given the text for a turn, split the string into a prefix containing this turn/speaker information (eg. "379: Bob: "), and a body containing the turn text itself (eg. "I think it's your go, Alice").

Mind your offsets! They're based on the whole turn string.

educe.stac.annotation.**split_type**(*anno*)
An object's type as a (frozen)set of items. You're probably looking for *educe.stac.dialogue_act* instead.

educe.stac.annotation.**turn_id**(*anno*)
Get the turn identifier for a turn annotation (or None).

Parameters **anno** (Annotation) – Annotation

Returns **turn_id** – Turn identifier ; None if the annotation has no feature 'Identifier'.

Return type tuple(int) or None

educe.stac.annotation.**twin**(*corpus*, *anno*, *stage='units'*)
Given an annotation in a corpus, retrieve the equivalent annotation (by local identifier) from a a different stage of the corpus. Return this "twin" annotation or None if it is not found

Note that the annotation's origin must be set

The typical use of this would be if you have an EDU in the 'discourse' stage and need to get its 'units' stage equvialent to have its dialogue act.

Parameters **twin_doc** – unit-level document to fish twin from (None if you want educe to search for it in the corpus; NB: corpus can be None if you supply this)

educe.stac.annotation.**twin_from**(*doc*, *anno*)
Given a document and an annotation, return the first annotation in the document with a matching local identifier.

educe.stac.context module

The dialogue and turn surrounding an EDU along with some convenient information about it

class educe.stac.context.**Context**(*turn*, *tstar*, *turn_edus*, *dialogue*, *dialogue_turns*, *doc_turns*, *tokens=None*)
Bases: object

Representation of the surrounding context for an EDU, basically the relevant enclosing annotations: turns, dialogues. The idea is potentially extend this to a somewhat richer notion of context, including things like a sentence count, etc.

Parameters

- **turn** – the turn surrounding this EDU
- **tstar** – the tstar turn surrounding this EDU (a tstar turn is a sort of virtual turn made by merging consecutive turns in a dialogue that have the same speaker)
- **turn_edus** – the EDUs in the this turn
- **dialogue** – the dialogue surrounding this EDU
- **dialogue_turns** – all the turns in the dialogue surrounding this EDU (non-empty, sorted by first-widest span)
- **doc_turns** – all the turns in the document
- **tokens** – (may not be present): tokens contained within this EDU

classmethod **for_edus**(*doc*, *postags=None*)
Get a dictionary of context objects for each EDU in the doc.

Returns **contexts** – A dictionary with a context for each EDU in the document.

Return type dict(educe.glozz.Unit, *Context*)

speaker()
the speaker associated with the turn surrounding an edu

educe.stac.context.**containing**(*span*, *annos*)
Given an iterable of standoff, pick just those that enclose/contain the given span (ie. are bigger and around)

educe.stac.context.**edus_in_span**(*doc*, *span*)
Given an document and a text span return the EDUs the document contains in that span

educe.stac.context.**enclosed**(*span*, *annos*)
Given an iterable of standoff, pick just those that are enclosed by the given span (ie. are smaller and within)

educe.stac.context.**merge_turn_stars**(*doc*)
Return a copy of the document in which consecutive turns by the same speaker have been merged.

Merging is done by taking the first turn in grouping of consecutive speaker turns, and stretching its span over all the subsequent turns.

Additionally turn prefix text (containing turn numbers and speakers) from the removed turns are stripped out.

educe.stac.context.**sorted_first_widest**(*nodes*)
Given a list of nodes, return the nodes ordered by their starting point, and in case of a tie their inverse width (ie. widest first).

educe.stac.context.**speakers**(*contexts*, *anno*)
Return a list of speakers of an EDU or CDU (in the textual order of the EDUs).

educe.stac.context.**turns_in_span**(*doc*, *span*)
Given a document and a text span, return the turns that the document contains in that span

educe.stac.corenlp module

STAC conventions for running the Stanford CoreNLP pipeline, saving the results, and reading them.

The most useful functions here are

- run_pipeline
- read_results

educe.stac.corenlp.**from_corenlp_output_filename**(*f*)
Return a tuple of FileId and turn id.

This is entirely by convention we established when calling corenlp of course

educe.stac.corenlp.**parsed_file_name**(*k*, *dir_name*)
Given an educe.corpus.FileId and directory, return the file path within that directory that corresponds to the corenlp output

educe.stac.corenlp.**read_corenlp_result**(*doc*, *corenlp_doc*, *tid=None*)
Read CoreNLP's output for a document.

Parameters

- **doc** (*educe Document (?)*) – The original document (?)
- **corenlp_doc** (educe.external.stanford_xml_reader.PreprocessingSource) – Object that contains all annotations for the document
- **tid** (*turn id*) – Turn id (?)

Returns **corenlp_doc** – A CoreNlpDocument containing all information.

Return type *CoreNlpDocument*

educe.stac.corenlp.**read_results**(*corpus*, *dir_name*)
Read stored parser output from a directory, and convert them to educe.annotation.Standoff objects.

Return a dictionary mapping 'FileId's to sets of tokens.

educe.stac.corenlp.**run_pipeline**(*corpus*, *outdir*, *corenlp_dir*, *split=False*)
Run the standard corenlp pipeline on all the (unannotated) documents in the corpus and save the results in the specified directory.

If *split=True*, we output one file per turn, an experimental mode to account for switching between multiple speakers. We don't have all the infrastructure to read these back in (it should just be a matter of some filename manipulation though) and hope to flesh this out later. We also intend to tweak the notion of splitting by aggregating consecutive turns with the same speaker, which may somewhat mitigate the loss of coreference information.

`educe.stac.corenlp.`**`turn_id_text`**(*doc*)
Return a list of (turn ids, text) tuples in span order (no speaker)

educe.stac.corpus module

Corpus layout conventions (re-exported by educe.stac)

class `educe.stac.corpus.`**`LiveInputReader`**(*corpusdir*)
Bases: *educe.stac.corpus.Reader*

Reader for unannotated 'live' data that we want to parse.

The data is assumed to be in a directory with one aa/ac file pair.

There is no notion of subdocument (*subdoc = None*) and the stage is *'unannotated'*

`files`(*doc_glob=None*)

Parameters **`doc_glob`** (*str, optional*) – Glob expression for document (folder) names ; if *None*, it uses the wildcard '*' for file basenames.

class `educe.stac.corpus.`**`Reader`**(*corpusdir*)
Bases: *educe.corpus.Reader*

See *educe.corpus.Reader* for details

`files`(*doc_glob=None*)
Gather files for docs whose folder name matches *doc_glob*.

Parameters **`doc_glob`** (*str, optional*) – Glob expression for document (folder) names ; if *None*, it uses the wildcard '*' to match all strings.

`slurp_subcorpus`(*cfiles*, *verbose=False*)

`educe.stac.corpus.`**`id_to_path`**(*k*)
Given a fleshed out FileId (none of the fields are None), return a filepath for it following STAC conventions.

You will likely want to add your own filename extensions to this path

`educe.stac.corpus.`**`is_metal`**(*fileid*)
If the annotator is one of the distinguished standard annotators

`educe.stac.corpus.`**`twin_key`**(*key*, *stage*)
Given an annotation key, return a copy shifted over to a different stage.

Note that copying from unannotated to another stage, you will need to set the annotator

`educe.stac.corpus.`**`write_annotation_file`**(*anno_filename*, *doc*)
Write a GlozzDocument to XML in the given path

educe.stac.fake_graph module

Fake graphs for testing STAC algorithms

Specification for mini-language

Source string is parsed line by line, data type depends on first character Uppercase letters are speakers, lowercase letters are units EDU names are arranged following alphabetical order (does NOT apply to CDUs) Please arrange the lines in that order:

- # : speaker line

```
# Aabce Bdg Cfh
```

- any lowercase : CDU line (top-level last)

```
y(eg) x(wyz)
```

- S or C : relation line

```
Sabd bf ceCh
```

anything else : skip as comment

class `educe.stac.fake_graph.`**`LightGraph`**(*src*)
Structure holding only relevant information

Unit keys (sortable, hashable) must correspond to reading order CDUs can be placed in any position wrt their components

`get_doc`()

`get_edge`(*source*, *target*)
Return an educe.annotation.Relation for the given LightGraph names for source and target

`get_node`(*name*)
Return an educe.annotation.Unit or Schema for the given LightGraph name

educe.stac.fusion module

Somewhat higher level representation of STAC documents than the usual Glozz layer.

Note that this is a relatively recent addition to Educe. Up to the time of this writing (2015-03), we had two options for dealing with STAC:

- manually manipulating glozz objects via educe.annotation
- dealing with some high-level but not particularly helpful hypergraph objects

We try to provide an intermediary in this layer by merging information from several layers in one place.

A typical example might be to print a listing of

(edu1_id, edu2_id, edu1_dialogue_act, edu2_dialogue_act, relation_label)

This has always been a bit awkward when dealing with Glozz, because there are separate annotations in different Glozz documents, the dialogue acts in the 'units' stage; and the linked units in the discourse stage. Combining these streams has always involved a certain amount of manual lookup, which we hope to avoid with this fusion layer.

At the time of this writing, this will have a bit of emphasis on feature extraction.

class `educe.stac.fusion.`**`Dialogue`**(*anno*, *edus*, *relations*)
Bases: `object`

STAC Dialogue.

Note that input EDUs should be sorted by span.

edu_pairs()
Generate all EDU pairs within this dialogue.

This includes pairs whose source is the left padding (fake root) EDU.

Yields (source, target) (*tuple of educe.stac.annotation.Unit*) – Next candidate edge, as a pair of EDUs (source, target).

class educe.stac.fusion.**EDU**(*doc*, *discourse_anno*, *unit_anno*)
Bases: *educe.annotation.Unit*

STAC EDU

A STAC EDU merges information from the unit and discourse annotation stages so that you can ignore the distinction between the two annotation stages.

It also tries to be usable as a drop-in substitute for both annotations and contexts

dialogue_act()
The (normalised) speech act associated with this EDU (None if unknown)

fleshout(*context*)
second phase of EDU initialisation; fill out contextual info

identifier()
Some kind of identifier string that uniquely identfies the EDU in the corpus. Because these are higher level annotations than in the Glozz layer we will use the 'local' identifier, which should be the same across stages

is_left_padding()
If this is a virtual EDU used in machine learning tasks

speaker()
the speaker associated with the turn surrounding an edu

subgrouping()
What abstract subgrouping the EDU is in (here: turn stars)

See also:

educe.stac.context.merge_turn_stars()

Returns subgrouping

Return type string

text()
The text for just this EDU

educe.stac.fusion.**ROOT = 'ROOT'**
distinguished fake EDU id for machine learning applications

educe.stac.fusion.**fuse_edus**(*discourse_doc*, *unit_doc*, *postags*)
Return a copy of the discourse level doc, merging info from both the discourse and units stage.

All EDUs will be converted to higher level EDUs.

Notes

- The discourse stage is primary in that we work by going over what EDUs we find in the discourse stage and trying to enhance them with information we find on their units-level equivalents. Sometimes (rarely but it happens) annotations can go out of synch. EDUs missing on the units stage will be silently ignored

(we try to make do without them). EDUs that were introduced on the units stage but not percolated to discourse will also be ignored.

- We rely on annotation ids to match EDUs from both stages; it's up to you to ensure that the annotations are really in synch.
- This does not constitute a full merge of the documents. For a full merge, you would have to bring over other annotations such as Resources, *Preference*, *Anaphor*, *Several_resources*, taking care all the while to ensure there are no timestamp clashes with pre-existing annotations (it's unlikely but best be on the safe side if you ever find yourself with automatically generated annotations, where all bets are off time-stamp wise).

Parameters

- **discourse_doc** (GlozzDocument) – Document from the "discourse" stage.
- **unit_doc** (GlozzDocument) – Document from the "units" stage.
- **postags** (*list of Token*) – Sequence of educe tokens predicted by the POS tagger for this document.

Returns **doc** – Deep copy of the discourse_doc with info from the units stage merged in.

Return type *GlozzDocument*

educe.stac.graph module

STAC-specific conventions related to graphs.

class educe.stac.graph.**DotGraph**(*anno_graph*)
Bases: *educe.graph.DotGraph*

A dot representation of this graph for visualisation. The *to_string()* method is most likely to be of interest here

class educe.stac.graph.**EnclosureDotGraph**(*core*)
Bases: *educe.graph.EnclosureDotGraph*

Conventions for visualising STAC enclosure graphs

class educe.stac.graph.**EnclosureGraph**(*doc*, *postags=None*)
Bases: *educe.graph.EnclosureGraph*

An enclosure graph based on STAC conventions

class educe.stac.graph.**Graph**
Bases: *educe.graph.Graph*

cdu_head(*cdu*, *sloppy=False*)
Get the head DU of a CDU.

The head of a CDU is defined here as the only DU that is not pointed to by any other member of this CDU.

This is meant to approximate the description in (Muller 2012) (/Constrained decoding for text-level discourse parsing/):

1. in the highest DU in its subgraph in terms of suboordinate relations,
2. in case of a tie in #1, the leftmost in terms of coordinate relations.

Corner cases:

- Return None if the CDU has no members (annotation error)

- If the CDU contains more than one head (annotation error) and if sloppy is True, return the textually leftmost one; otherwise, raise a MultiheadedCduException

Parameters

- **cdu** (CDU) – The CDU under examination.
- **sloppy** (*boolean, defaults to False*) – If True, return the textually leftmost DU if the CDU contains more than one head ; if False, raise a *MultiheadedCduException* in such cases.

Returns **cand** – The head DU of this CDU ; it is None if no member of the CDU qualifies as a head (loop?).

Return type *Unit* or Schema? or None

first_outermost_dus()
Return discourse units in this graph, ordered by their starting point, and in case of a tie their inverse width (ie. widest first)

classmethod from_doc(*corpus*, *doc_key*, *pred=<function <lambda>>*)

is_cdu(*x*)

is_edu(*x*)

is_relation(*x*)

recursive_cdu_heads(*sloppy=False*)
A dictionary mapping each CDU to its recursive CDU head (see *cdu_head*)

sorted_first_outermost(*annos*)
Order nodes by their starting point, then inverse width.

Given a list of nodes, return the nodes ordered by their starting point, and in case of a tie their inverse width (ie. widest first).

strip_cdus(*sloppy=False*, *mode='head'*)
Delete all CDUs in this graph.

Links involving a CDU will point to/from the elements of this CDU. Non-head modes may add new edges to the graph.

Parameters

- **sloppy** (*boolean, default=False*) – See *cdu_head*.
- **mode** (*string, default='head'*) – Strategy for replacing edges involving CDUs. *head* will relocate the edge on the recursive head of the CDU (see *recursive_cdu_heads*). *broadcast* will distribute the edge over all EDUs belonging to the CDU. A copy of the edge will be created for each of them. If the edge's source and target are both distributed, a new copy will be created for each combination of EDUs. *custom* (or any other string) will distribute or relocate on the head depending on the relation label.

without_cdus(*sloppy=False*, *mode='head'*)
Return a deep copy of this graph with all CDUs removed. Links involving these CDUs will point instead from/to their deep heads

We'll probably deprecate this function, since you could just as easily call deepcopy yourself

exception educe.stac.graph.**MultiheadedCduException**(*cdu*, **args*, ***kw*)
Bases: exceptions.Exception

class educe.stac.graph.**WrappedToken**(*token*)
Bases: *educe.annotation.Annotation*

Thin wrapper around POS tagged token which adds a local_id field for use by the EnclosureGraph mechanism

educe.stac.postag module

STAC conventions for running a pos tagger, saving the results, and reading them.

educe.stac.postag.**extract_turns**(*doc*)
Return a string representation of the document's turn text for use by a tagger

educe.stac.postag.**read_tags**(*corpus*, *root_dir*)
Read stored POS tagger output from a directory, and convert them to educe.annotation.Standoff objects.

Return a dictionary mapping 'FileId's to sets of tokens.

Parameters

- **corpus** (*dict(FileId, GlozzDocument)*) – Dictionary of documents keyed by their FileId.
- **root_dir** (*str*) – Path to the directory containing the output of the POS tagger, one file per document.

Returns **pos_tags** – Map from each document id to the list of tokens predicted by a POS tagger.

Return type dict(*FileId*, list(*Token*))

educe.stac.postag.**run_tagger**(*corpus*, *outdir*, *tagger_jar*)
Run the ark-tweet-tagger on all the (unannotated) documents in the corpus and save the results in the specified directory

educe.stac.postag.**sorted_by_span**(*annos*)
Annotations sorted by text span

educe.stac.postag.**tagger_cmd**(*tagger_jar*, *txt_file*)
Command to run the POS tagger

educe.stac.postag.**tagger_file_name**(*doc_key*, *root*)
Get the file path to the output of the POS tagger for a document.

The returned file path is a .conll file within the given directory.

Parameters

- **doc_key** (educe.corpus.FileId) – FileId of the document
- **root** (*string*) – Path to the folder containing annotations for this corpus, including the output of the POS tagger.

Returns **res** – Path to the .conll file output by the POS tagger.

Return type string

educe.stac.rfc module

Right frontier constraint and its variants

class educe.stac.rfc.**BasicRfc**(*graph*)
Bases: object

The vanilla right frontier constraint

```
1. X is textually last => RF(X)

2. Y
   | (sub)
   v
   X

   RF(Y) => RF(X)

3. X: +----+
      | Y  |
      +----+

   RF(Y) => RF(X)
```

frontier()
Return the list of nodes on the right frontier of the whole graph

violations()
Return a list of relation instance names, corresponding to the RF violations for the given graph.

You'll need a stac graph object to interpret these names with.

Return type [string]

class educe.stac.rfc.**ThreadedRfc**(*graph*)
Bases: *educe.stac.rfc.BasicRfc*

Same as BasicRfc except for point 1:

1. X is the textual last utterance of any speaker => RF(X)

educe.stac.rfc.**powerset**(*[1,2,3]) –> () (1,) (2,) (3,) (1,2) (1,3) (2,3) (1,2,3*)

educe.stac.rfc.**speakers**(*contexts*, *anno*)
Returns the speakers for given annotation unit

Takes : contexts (Context dict), Annotation

4.4 Submodules

4.5 educe.annotation module

Low-level representation of corpus annotations, following somewhat faithfully the Glozz model for annotations.

This is low-level in the sense that we make little attempt to interpret the information stored in these annotations. For example, a relation might claim to link two units of id unit42 and unit43. This being a low-level representation, we simply note the fact. A higher-level representation might attempt to actually make the corresponding units available to you, or perhaps provide some sort of graph representation of them

class educe.annotation.**Annotation**(*anno_id*, *span*, *atype*, *features*, *metadata=None*, *origin=None*)
Bases: *educe.annotation.Standoff*

Any sort of annotation.

Annotations tend to have: * span: some sort of location (what they are annotating) * type: some key label (we call a type) * features: an attribute to value dictionary

identifier()
Global identifier if possible, else local identifier.

String representation of an identifier that should be unique to this corpus at least.

If the unit has an origin (see "FileId"), we use the

- document
- subdocument
- stage
- (but not the annotator!)
- and the id from the XML file

If we don't have an origin we fall back to just the id provided by the XML file.

See also *position* as potentially a safer alternative to this (and what we mean by safer)

local_id()
Local identifier.

An identifier which is sufficient to pick out this annotation within a single annotation file.

class educe.annotation.**Document**(*units*, *relations*, *schemas*, *text*)
Bases: *educe.annotation.Standoff*

A single (sub)-document.

This can be seen as collections of unit, relation, and schema annotations

annotations()
All annotations associated with this document

fleshout(*origin*)
See *set_origin*

global_id(*local_id*)
String representation of an identifier that should be unique to this corpus at least.

set_origin(*origin*)
If you have more than one document, it's a good idea to set its origin to a file ID so that you can more reliably the annotations apart.

text(*span=None*)
Return the text associated with these annotations (or None), optionally limited to a span

class educe.annotation.**RelSpan**(*t1*, *t2*)
Bases: object

Which two units a relation connects.

t1 = None
string – id of an annotation

t2 = None
string – id of an annotation

class educe.annotation.**Relation**(*rel_id*, *span*, *rtype*, *features*, *metadata=None*)
Bases: *educe.annotation.Annotation*

An annotation between two annotations.

Relations are directed; see *RelSpan* for details

Use the *source* and *target* field to grab these respective annotations, but note that they are only instantiated after *fleshout* is called (corpus slurping normally fleshes out documents and thus their relations).

fleshout(*objects*)
Given a dictionary mapping ids to annotation objects, set this relation's source and target fields.

source = None
source annotation; will be defined by fleshout

target = None
target annotation; will be defined by fleshout

class educe.annotation.**Schema**(*rel_id*, *units*, *relations*, *schemas*, *stype*, *features*, *metadata=None*)
Bases: *educe.annotation.Annotation*

An annotation between a set of annotations

Use the *members* field to grab the annotations themselves. But note that it is only created when *fleshout* is called.

fleshout(*objects*)
Given a dictionary mapping ids to annotation objects, set this schema's *members* field to point to the appropriate objects

terminals()
All unit-level annotations contained in this schema or (recursively in schema contained herein)

class educe.annotation.**Span**(*start*, *end*)
Bases: object

What portion of text an annotation corresponds to. Assumed to be in terms of character offsets

The way we interpret spans in educe amounts to how Python interprets array slice indices.

One way to understand them is to think of offsets as sitting in between individual characters

```
  h   o   w   d   y
0   1   2   3   4   5
```

So *(0,5)* covers the whole word above, and *(1,2)* picks out the letter "o"

absolute(*other*)
Assuming this span is relative to some other span, return a suitably shifted "absolute" copy.

encloses(*other*)
Return True if this span includes the argument

Note that *x.encloses(x) == True*

Corner case: *x.encloses(None) == False*

See also *educe.graph.EnclosureGraph* if you might be repeating these checks

length()
Return the length of this span

merge(*other*)
Return a span that stretches from the beginning to the end of the two spans. Whereas *overlaps* can be thought of as returning the intersection of two spans, this can be thought of as returning the union.

classmethod merge_all(*spans*)
Return a span that stretches from the beginning to the end of all the spans in the list

overlaps(*other*, *inclusive=False*)
Return the overlapping region if two spans have regions in common, or else None.

```
Span(5, 10).overlaps(Span(8, 12)) == Span(8, 10)
Span(5, 10).overlaps(Span(11, 12)) == None
```

If *inclusive* == *True*, spans with touching edges are considered to overlap

```
Span(5, 10).overlaps(Span(10, 12)) == None
Span(5, 10).overlaps(Span(10, 12), inclusive=True) == Span(10, 10)
```

relative(*other*)
Assuming this span is relative to some other span, return a suitably shifted "absolute" copy.

shift(*offset*)
Return a copy of this span, shifted to the right (if offset is positive) or left (if negative).

It may be a bit more convenient to use 'absolute/relative' if you're trying to work with spans that are within other spans.

class educe.annotation.**Standoff**(*origin=None*)
Bases: object

A standoff object ultimately points to some piece of text.

The pointing is not necessarily direct though.

origin
educe.corpus.FileId, optional – FileId of the document supporting this standoff.

encloses(*other*)
True if this annotation's span encloses the span of the other.

s1.encloses(s2) is shorthand for *s1.text_span().encloses(s2.text_span())*

Parameters **other** (Standoff) – Other annotation.

Returns **res** – True if this annotation's span encloses the span of the other.

Return type boolean

overlaps(*other*)
True if this annotations's span overlaps with the span of the other.

s1.overlaps(s2) is shorthand for *s1.text_span().overlaps(s2.text_span())*

Parameters **other** (Standoff) – Other annotation.

Returns **res** – True if this annotation's span overlaps with the span of the other.

Return type boolean

text_span()
Return the span from the earliest terminal annotation contained here to the latest.

Corner case: if this is an empty non-terminal (which would be a very weird thing indeed), return None.

Returns **res** – Span from the first character of the earliest terminal annotation contained here, to the last character of the latest terminal annotation ; None if this annotation has no terminal.

Return type *Span* or None

class educe.annotation.**Unit**(*unit_id*, *span*, *utype*, *features*, *metadata=None*, *origin=None*)
Bases: *educe.annotation.Annotation*

Unit annotation.

An annotation over a span of text.

position()
The position is the set of "geographical" information only to identify an item. So instead of relying on some sort of name, we might rely on its text span. We assume that some name-based elements (document name, subdocument name, stage) can double as being positional.

If the unit has an origin (see "FileId"), we use the

- document
- subdocument
- stage
- (but not the annotator!)
- and its text span

position vs identifier

This is a trade-off. On the one hand, you can see the position as being a safer way to identify a unit, because it obviates having to worry about your naming mechanism guaranteeing stability across the board (eg. two annotators stick an annotation in the same place; does it have the same name). On the *other* hand, it's a bit harder to uniquely identify objects that may coincidentally fall in the same span. So how much do you trust your IDs?

4.6 educe.corpus module

Corpus management

class educe.corpus.**FileId**(*doc*, *subdoc*, *stage*, *annotator*)
Information needed to uniquely identify an annotation file.

Note that this includes the annotator, so if you want to do comparisons on the "same" file between annotators you'll want to ignore this field.

Parameters

- **doc** (*string*) – document name
- **subdoc** (*string*) – subdocument (often None); sometimes you may have a need to divide a document into smaller pieces (for exmaple working with tools that require too much memory to process large documents). The subdocument identifies which piece of the document you are working with. If you don't have a notion of subdocuments, just use None
- **stage** (*string*) – annotation stage; for use if you have distinct files that correspond to different stages of your annotation process (or different processing tools)
- **annotator** (*string*) – the annotator (or annotation tool) that generated this annoation file

mk_global_id(*local_id*)
String representation of an identifier that should be unique to this corpus at least.

If the unit has an origin (see "FileId"), we use the

- document
- subdocument
- (but not the stage!)

- (but not the annotator!)
- and the id from the XML file

If we don't have an origin we fall back to just the id provided by the XML file

See also *position* as potentially a safer alternative to this (and what we mean by safer)

class `educe.corpus.`**`Reader`**(*root*)

Reader provides little more than dictionaries from *FileId* to data.

Parameters **`rootdir`** (*str*) – the top directory of the corpus

A potentially useful pattern to apply here is to take a slice of these dictionaries for processing. For example, you might not want to read the whole corpus, but only the files which are modified by certain annotators.

```
reader = Reader(corpus_dir)
files = reader.files()
subfiles = {k: v in files.items() if k.annotator in ['Bob', 'Alice']}
corpus = reader.slurp(subfiles)
```

Alternatively, having read in the entire corpus, you might be doing processing on various slices of it at a time

```
corpus = reader.slurp()
subcorpus = {k: v in corpus.items() if k.doc == 'pilot14'}
```

This is an abstract class; you should use the version from a data-set, eg. *educe.stac.Reader* instead

`files`(*doc_glob=None*)

Return a dictionary from FileId to (tuples of) filepaths. The tuples correspond to files that are considered to 'belong' together; for example, in the case of standoff annotation, both the text file and its annotations

Derived classes

Parameters **`doc_glob`** (*str, optional*) – Glob expression for names of game folders ; if *None*, subclasses are expected to use the wildcard '*' that matches all strings.

`filter`(*d*, *pred*)

Convenience function equivalent to

```
{ k:v for k,v in d.items() if pred(k) }
```

`slurp`(*cfiles=None*, *doc_glob=None*, *verbose=False*)

Read the entire corpus if *cfiles* is *None* or else the subset specified by *cfiles*.

Return a dictionary from FileId to *educe.Annotation.Document*

Parameters

- **`cfiles`** (*dict, optional*) – Dict of files like what *Corpus.files()* would return.
- **`doc_glob`** (*str, optional*) – Glob pattern for doc (folder) names ; ignored if *cfiles* is not None.
- **`verbose`** (*boolean, defaults to False*) – If True, print what we're reading to stderr.

`slurp_subcorpus`(*cfiles*, *verbose=False*)

Derived classes should implement this function

4.7 educe.glozz module

The Glozz file format in *educe.annotation* form

You're likely most interested in *slurp_corpus* and *read_annotation_file*

class educe.glozz.**GlozzDocument**(*hashcode*, *unit*, *rels*, *schemas*, *text*)
Bases: *educe.annotation.Document*

Representation of a glozz document

set_origin(*origin*)

to_xml(*settings=<educe.glozz.GlozzOutputSettings object>*)

exception educe.glozz.**GlozzException**(**args*, ***kw*)
Bases: exceptions.Exception

class educe.glozz.**GlozzOutputSettings**(*feature_order*, *metadata_order*)
Bases: object

Non-essential aspects of Glozz XML output, such as the order that feature structures or metadata are written out. Controlling these settings could be useful when you want to automatically modify an existing Glozz document, but produce only minimal textual diffs along the way for revision control, comparability, etc.

educe.glozz.**glozz_annotation_to_xml**(*self*, *tag='annotation'*, *settings=<educe.glozz.GlozzOutputSettings object>*)

educe.glozz.**glozz_relation_to_span_xml**(*self*)

educe.glozz.**glozz_schema_to_span_xml**(*self*)

educe.glozz.**glozz_unit_to_span_xml**(*self*)

educe.glozz.**hashcode**(*f*)
Hashcode mechanism as documented in the Glozz manual appendix. Hint, using cStringIO to get the hashcode for a string

educe.glozz.**ordered_keys**(*preferred*, *d*)
Keys from a dictionary starting with 'preferred' ones in the order of preference

educe.glozz.**read_annotation_file**(*anno_filename*, *text_filename=None*)
Read a single glozz annotation file and its corresponding text (if any).

educe.glozz.**read_node**(*node*, *context=None*)

educe.glozz.**write_annotation_file**(*anno_filename*, *doc*, *settings=<educe.glozz.GlozzOutputSettings object>*)
Write a GlozzDocument to XML in the given path

4.8 educe.graph module

Graph representation of discourse structure. Classes of interest:

- Graph: the core structure, use the *Graph.from_doc* factory method to build one out of an *educe.annotation* document.
- DotGraph: visual representation, built from *Graph*. You probably want a project-specific variant to get more helpful graphs, see eg. *educe.stac.Graph.DotGraph*

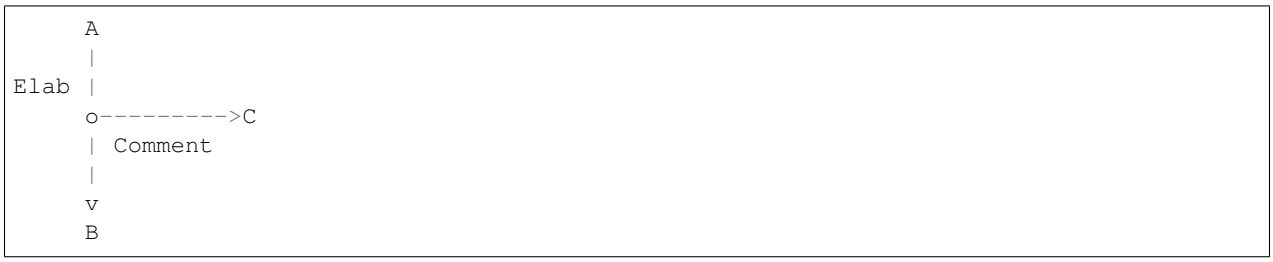
4.8.1 Educe hypergraphs

Somewhat tricky hypergraph representation of discourse structure.

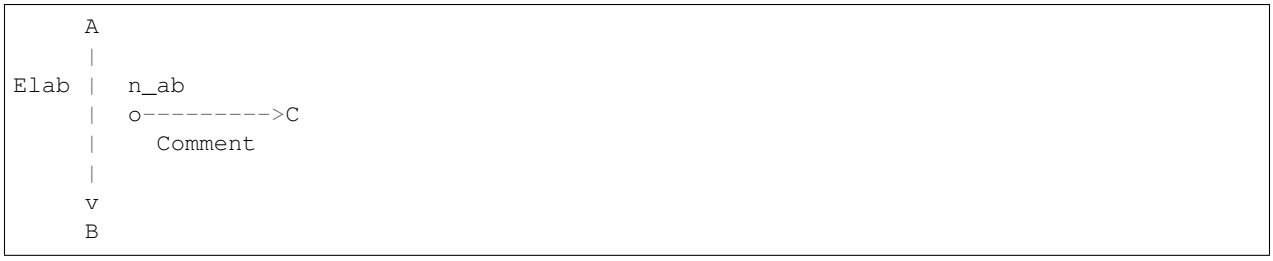
- a node for every elementary discourse unit
- a hyperedge for every relation instance[1]
- a hyperedge for every complex discourse unit
- (the tricky bit) for every (hyper)edge *e_x* in the graph, introduce a "mirror node" *n_x* for that edge (this node also has *e_x* as its "mirror edge")

The tricky bit is a response to two issues that arise: (A) how do we point to a CDU? Our hypergraph formalism and library doesn't have a notion of pointing to hyperedges (only nodes) and (B) what do we do about misannotations where we have relation instances pointing to relation instances? *A* is the most important one to address (in principle, we could just treat *B* as an error and raise an exception), but for now we decide to model both scenarios, and the same "mirror" mechanism above.

The mirrors are a bit problematic because are not part of the formal graph structure (think of them as extra labels). This could lead to some seriously unintuitive consequences when traversing the graph. For example, if you two DUs A and B connected by an Elab instance, and if that instance is itself (bizarrely) connected to some other DU, you might intuitively expect A, B, and C to all form one connected component



Alas, this is not so! The reality is a bit messier, with there being no formal relationship between edge and mirror



The same goes for the connectedness of things pointing to CDUs and with their members. Looking at pictures, you might intuitively think that if a discourse unit (A) were connected to a CDU, it would also be connected to the discourse units within

[1] just a binary hyperedge, ie. like an edge in a regular graph. As these are undirected, we take the convention that the the first link is the tail (from) and the second link is the tail (to).

The reality is messier for the same reasons above

```
     A
     |
Elab |      +-----+ e_bc
     |      | B C |
     v      +-----+
     n_bc
```

4.8.2 Classes

class educe.graph.**AttrsMixin**
Attributes common to both the hypergraph and directed graph representation of discourse structure

annotation(*x*)
Return the annotation object corresponding to a node or edge

edge_attributes_dict(*x*)

edgeform(*x*)
Return the argument if it is an edge id, or its mirror if it's an edge id

(This is possible because every edge in the graph has a node that corresponds to it)

is_cdu(*x*)

is_edu(*x*)

is_relation(*x*)

mirror(*x*)
For objects (particularly, relations/CDUs) that have a mirror image, ie. an edge representation if it's a node or vice-versa, return the identifier for that image

node(*x*)
DEPRECATED (renamed 2013-11-19): use *self.nodeform(x)* instead

node_attributes_dict(*x*)

nodeform(*x*)
Return the argument if it is a node id, or its mirror if it's an edge id

(This is possible because every edge in the graph has a node that corresponds to it)

type(*x*)
Return if a node/edge is of type 'EDU', 'rel', or 'CDU'

class educe.graph.**DotGraph**(*anno_graph*)
Bases: pydot.Dot

A dot representation of this graph for visualisation. The *to_string()* method is most likely to be of interest here

This is fairly abstract and unhelpful. You probably want the project-layer extension instead, eg. *educe.stac.graph*

exception educe.graph.**DuplicateIdException**(*duplicate*)
Bases: exceptions.Exception

Condition that arises in inconsistent corpora

class educe.graph.**EnclosureDotGraph**(*enc_graph*)
Bases: pydot.Dot

class educe.graph.**EnclosureGraph**(*annotations*, *key=None*)
Bases: pygraph.classes.digraph.digraph, *educe.graph.AttrsMixin*

Caching mechanism for span enclosure. Given an iterable of Annotation, return a directed graph where nodes point to the largest nodes they enclose (i.e. not to nodes that are enclosed by intermediary nodes they point to). As a slight twist, we also allow nodes to redundantly point to enclosed nodes of the same typ.

This *should* give you a multipartite graph with each layer representing a different type of annotation, but no promises! We can't guarantee that the graph will be nicely layered because the annotations may be buggy (either nodes wrongly typed, or nodes of the same type that wrongly enclose each other), so you should not rely on this property aside from treating it as an optimisation.

Note: there is a corner case for nodes that have the same span. Technically a span encloses itself, so the graph could have a loop. If you supply a sort key that differentiates two nodes, we use it as a tie-breaker (first node encloses second). Otherwise, we simply exclude both links.

NB: nodes are labelled by their annotation id

Initialisation parameters

- annotations - iterable of Annotation
- **key - disambiguation key for nodes with same span** (annotation -> sort key)

inside(*annotation*)
Given an annotation, return all annotations that are directly within it. Results are returned in the order of their local id

outside(*annotation*)
Given an annotation, return all annotations it is directly enclosed in. Results are returned in the order of their local id

class educe.graph.**Graph**
Bases: pygraph.classes.hypergraph.hypergraph, *educe.graph.AttrsMixin*

Hypergraph representation of discourse structure. See the section on Educe *hypergraphs*

You most likely want to use *Graph.from_doc* instead of instantiating an instance directly

Every node/hyperedge is represented as string unique within the graph. Given one of these identifiers *x* and a graph *g*:

- *g.type(x)* returns one of the strings "EDU", "CDU", "rel"
- *g.annotation(x)* returns an educe.annotation object
- for relations and CDUs, if *e_x* is the edge representation of the relation/cdu, *g.mirror(x)* will return its mirror node *n_x* and vice-versa

TODOS:

- TODO: Currently we use educe.annotation objects to represent the EDUs, CDUs and relations, but this is likely a bit too low-level to be helpful. It may be nice to have higher-level EDU and CDU objects instead

cdu_members(*cdu*, *deep=False*)
Return the set of EDUs, CDUs, and relations which can be considered as members of this CDU.

This is shallow by default, in that we only return the immediate members of the CDU. If *deep==True*, also return members of CDUs that are members of (members of ..) this CDU.

cdus()
Set of hyperedges representing complex discourse units.

See also *cdu_members*

connected_components()
Return a set of a connected components.

Each connected component set can be passed to *self.copy()* to be copied as a subgraph.

This builds on python-graph's version of a function with the same name but also adds awareness of our conventions about there being both a node/edge for relations/CDUs.

containing_cdu(*node*)
Given an EDU (or CDU, or relation instance), return immediate containing CDU (the hyperedge) if there is one or None otherwise. If there is more than one containing CDU, return one of them arbitrarily.

containing_cdu_chain(*node*)
Given an annotation, return a list which represents its containing CDU, the container's container, and forth. Return the empty list if no CDU contains this one.

copy(*nodeset=None*)
Return a copy of the graph, optionally restricted to a subset of EDUs and CDUs.

Note that if you include a CDU, then anything contained by that CDU will also be included.

You don't specify (or otherwise have control over) what relations are copied. The graph will include all hyperedges whose links are all (a) members of the subset or (b) (recursively) hyperedges included because of (a) and (b)

Note that any non-EDUs you include in the copy set will be silently ignored.

This is a shallow copy in the sense that the underlying layer of annotations and documents remains the same.

Parameters **nodeset** (*iterable of strings*) – only copy nodes with these names

edus()
Set of nodes representing elementary discourse units

classmethod **from_doc**(*corpus*, *doc_key*, *could_include=<function <lambda>>*, *pred=<function <lambda>>*)
Return a graph representation of a document

Note: check the project layer for a version of this function which may be more appropriate to your project

Parameters

- **corpus** (dict from *FileId* to documents) – educe corpus dictionary
- **doc_key** (*FileId*) – key pointing to the document
- **could_include** (*annotation -> boolean*) – predicate on unit level annotations that should be included regardless of whether or not we have links to them
- **pred** (*annotation -> boolean*) – predicate on annotations providing some requirement they must satisfy in order to be taken into account (you might say that *could_include* gives; and *pred* takes away)

rel_links(*edge*)
Given an edge in the graph, return a tuple of its source and target nodes.

If the edge has only a single link, we assume it's a loop and return the same value for both

relations()
Set of relation edges representing the relations in the graph. By convention, the first link is considered the source and the the second is considered the target.

4.9 educe.internalutil module

Utility functions which are meant to be used by educe but aren't expected to be too useful outside of it

exception `educe.internalutil.`**`EduceXmlException`**(**args*, ***kw*)
Bases: `exceptions.Exception`

`educe.internalutil.`**`indent_xml`**(*elem*, *level=0*)
From <http://effbot.org/zone/element-lib.htm>

WARNING: destructive

`educe.internalutil.`**`linebreak_xml`**(*elem*)
Insert a break after each element tag

You probably want *indent_xml* instead

`educe.internalutil.`**`on_single_element`**(*root*, *default*, *f*, *name*)
Return

- the default if no elements
- f(the node) if one element
- an exception if more than one

`educe.internalutil.`**`treenode`**(*tree*)
API-change padding for NLTK 2 vs NLTK 3 trees

4.10 educe.util module

Miscellaneous utility functions

`educe.util.`**`FILEID_FIELDS = ['stage', 'doc', 'subdoc', 'annotator']`**
String representation of fields recognised in an educe.corpus.FileId

`educe.util.`**`add_corpus_filters`**(*parser*, *fields=None*, *choice_fields=None*)
For help with script-building:

Augment an argparser with options to filter a corpus on the various attributes in a 'educe.corpus.FileId' (eg, document, annotator).

Parameters

- **`fields`** (*`[String]`*) – which flag names to include (defaults to *FILEID_FIELDS*)
- **`choice_fields`** (*`Dict String [String]`*) – fields which accept a limited range of answers

Meant to be used in conjunction with *mk_is_interesting*

`educe.util.`**`add_subcommand`**(*subparsers*, *module*)
Add a subcommand to an argparser following some conventions:

- the module can have an optional NAME constant (giving the name of the command); otherwise we assume it's the unqualified module name
- the first line of its docstring is its help text
- subsequent lines (if any) form its epilog

Returns the resulting subparser for the module

educe.util.**concat**(*items*)
:: Iterable (Iterable a) -> Iterable a

educe.util.**concat_l**(*items*)
:: [[a]] -> [a]

educe.util.**fields_without**(*unwanted*)
Fields for *add_corpus_filters* without the unwanted members

educe.util.**mk_is_interesting**(*args*, *preselected=None*)
Return a function that when given a FileId returns 'True' if the FileId would be considered interesting according to the arguments passed in.

Parameters **preselected** (*Dict String [String]*) – fields for which we already know what matches we want

Meant to be used in conjunction with *add_corpus_filters*

educe.util.**relative_indices**(*group_indices*, *reverse=False*, *valna=None*)
Generate a list of relative indices inside each group. Missing (None) values are handled specifically: each missing value is mapped to *valna*.

Parameters

- **reverse** (*boolean, optional*) – If True, compute indices relative to the end of each group.
- **valna** (*int or None, optional*) – Relative index for missing values.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Bibliography

[li2014text] Li, S., Wang, L., Cao, Z., & Li, W. (2014).

Python Module Index

e

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X