PyDelphin Documentation

Release 0.9.2

Michael Wayne Goodman

Jul 08, 2019

Guides:

Quick Links

- Project page
- How to contribute
- Report a bug
- Changelog
- Code of conduct
- License (MIT)

CHAPTER 1

Requirements, Installation, and Testing

PyDelphin releases are available on PyPI and the source code is on GitHub. For most users, the easiest and recommended way of installing PyDelphin is from PyPI via **pip**, as it installs any required dependencies and makes the **delphin** command available (see *PyDelphin at the Command Line*). If you wish to inspect or contribute code, or if you need the most up-to-date features, PyDelphin may be used directly from its source files.

1.1 Requirements

PyDelphin works with Python versions 2.7 and 3.4+, regardless of the platform. Certain features, however, may require additional dependencies or may be platform specific, as shown in the table below:

Module	Dependencies	Notes
`delphin.extra.highlight`	Pygments	
`delphin.extra.latex`	tikz-dependency	LaTeX package
`delphin.interfaces.ace`	ACE	Linux and Mac only
`delphin.interfaces.rest`	requests	
`delphin.mrs.compare`	NetworkX	
`delphin.mrs.penman`	Penman	

1.2 Installing from PyPI

Install the latest releast from PyPI using **pip**:

```
$ pip install pydelphin
```

If you already have PyDelphin installed, you can upgrade it by adding the `--upgrade` flag to the command.

Note: In general, it is a good idea to use virtual environments to keep Python packages and dependencies confined to a specific environment. For more information, see here: https://packaging.python.org/tutorials/installing-packages/

#creating-virtual-environments

1.3 Running from Source

Clone the repository from GitHub to get the latest source code:

```
$ git clone https://github.com/delph-in/pydelphin.git
```

By default, cloning the git repository checks out the `develop` branch. If you want to work in a difference branch (e.g., `master` for the code of the latest release), you'll need to `checkout` the branch (e.g., `$ git checkout master`).

In order to use PyDelphin from source, it will need to be importable by Python. If you are using PyDelphin as a library for your own project, you can adjust PYTHONPATH to point to PyDelphin's top directory, e.g.:

```
$ PYTHONPATH=~/path/to/pydelphin/ python myproject.py
```

Also note that the dependencies of any PyDelphin features you use will need to be satisfied manually.

Alternatively, **`pip`** can install PyDelphin from the source directory instead of from PyPI, and it will detect and install the dependencies:

```
$ pip install ~/path/to/pydelphin/
```

> **Warning:** The PyDelphin source code can be installed simply by running `$ setup.py install`, but this method is not recommended because uninstalling PyDelphin and its dependencies becomes more difficult.

1.4 Running Unit Tests

PyDelphin's unit tests are not distributed on PyPI, so if you wish to run the unit tests you'll need to get the source code. The tests are written for pytest, so one way to run them is by installing **`pytest`** (for both Python 2 and 3), and running:

```
$ pytest --doctest-glob=tests/\*.md
```

A better way to run the tests is using tox:

```
$ tox
```

The **`tox`** utility manages the building of virtual environments for testing, and it uses a configuration file (included in PyDelphin's sources) that specify how **`pytest`** should be called. It can also run the tests for several different Python versions. Note that simply running **`tox`** without any options will attempt to test PyDelphin with every supported version of Python, and it is likely that some of those versions are not installed on your system. The `-e` option to **`tox`** can specify the Python version; e.g., the following runs the tests for Python versions 2.7 and 3.6:

```
$ tox -e py27,py36
```

CHAPTER 2

Walkthrough of PyDelphin Features

This tutorial provides a tour of the main features offered by PyDelphin.

2.1 ACE and HTTP Interfaces

PyDelphin works with a number of data types, and a simple way to get some data to play with is to parse a sentence. PyDelphin doesn't parse things on its own, but it provides two interfaces to external processors: one for the ACE processor and another for the HTTP-based "web API". I'll first show the web API as it's the simplest for parsing a single sentence:

```
>>> from delphin.interfaces import rest
>>> response = rest.parse('Abrams chased Browne', params={'mrs': 'json'})
>>> response.result(0).mrs()
<Mrs object (proper named chase proper named) at 139897112151488>
```

The response object returned by interfaces is a basic dictionary that has been augmented with convenient access methods (such as `result()` and `mrs()` above). Note that the web API is platform-neutral, and is thus currently the only way to dynamically retrieve parses in PyDelphin on a Windows machine.

See also:

- Wiki for the HTTP ("RESTful") API: http://moin.delph-in.net/ErgApi
- Bottlenose server: https://github.com/delph-in/bottlenose
- *`delphin.interfaces.rest`* module
- `delphin.interfaces.ParseResponse` module

If you're on a Linux or Mac machine and have ACE installed and a grammar image available, you can use the ACE interface, which is faster than the web API and returns more complete response information.

```
>>> from delphin.interfaces import ace
>>> response = ace.parse('erg-1214-x86-64-0.9.27.dat', 'Abrams chased Browne')
NOTE: parsed 1 / 1 sentences, avg 2135k, time 0.01316s
```

(continues on next page)

(continued from previous page)

```
>>> response.result(0).mrs()
<Mrs object (proper named chase proper named) at 139897048034552>
```

See also:

- ACE: http://sweaglesw.org/linguistics/ace/
- *delphin.interfaces.ace* module
- *Using ACE from PyDelphin* tutorial

I will use the `response` object from ACE to illustrate some other features below.

2.2 *MRS Inspection

The original motivation for PyDelphin and the area with the most work is in modeling MRS representations.

```
>>> x = response.result(0).mrs()
>>> [ep.pred.string for ep in x.eps()]
['proper_q', 'named', '_chase_v_1', 'proper_q', 'named']
>>> x.variables()
['h0', 'e2', 'h4', 'x3', 'h5', 'h6', 'h7', 'h1', 'x9', 'h10', 'h11', 'h12', 'h13']
>>> x.nodeid('x3')
10001
>>> x.ep(x.nodeid('x3'))
<ElementaryPredication object (named (x3)) at 140597926475360>
>>> x.ep(x.nodeid('x3', quantifier=True))
<ElementaryPredication object (proper_q (x3)) at 140597926475240>
>>> x.ep(x.nodeid('e2')).args
{'ARG0': 'e2', 'ARG1': 'x3', 'ARG2': 'x9'}
>>> [(hc.hi, hc.relation, hc.lo) for hc in x.hcons()]
[('h0', 'qeq', 'h1'), ('h5', 'qeq', 'h7'), ('h11', 'qeq', 'h13')]
```

See also:

- Wiki of MRS topics: http://moin.delph-in.net/RmrsTop
- *delphin.mrs.xmrs* module

Beyond the basic modeling of semantic structures, there are a number of additional functions for analyzing the structures, such as from the *delphin.mrs.compare* and *delphin.mrs.query* modules:

```
>>> from delphin.mrs import compare
>>> compare.isomorphic(x, x)
True
>>> compare.isomorphic(x, response.result(1).mrs())
False
>>> from delphin.mrs import query
>>> query.select_eps(response.result(1).mrs(), pred='named')
[<ElementaryPredication object (named (x3)) at 140244783534752>,
↪<ElementaryPredication object (named (x9)) at 140244783534272>]
>>> x.args(10000)
{'ARG0': 'x3', 'RSTR': 'h5', 'BODY': 'h6'}
>>> query.find_argument_target(x, 10000, 'RSTR')
10001
>>> for sg in query.find_subgraphs_by_preds(x, ['named', 'proper_q'], connected=True):
...     print(sg.nodeids())
```

(continues on next page)

(continued from previous page)

```
...
[10000, 10001]
[10003, 10004]
```

See also:

- MRS isomorphism wiki: http://moin.delph-in.net/MrsIsomorphism
- *delphin.mrs.compare* module
- *delphin.mrs.query* module

2.3 *MRS Conversion

Conversions between MRS and DMRS representations is seamless in PyDelphin, making it easy to convert between many formats (note that some outputs are abbreviated here):

```
>>> from delphin.mrs import simplemrs, mrx, dmrx
>>> print(simplemrs.dumps([x], pretty_print=True))
[ TOP: h0
  INDEX: e2 [ e SF: prop TENSE: past MOOD: indicative PROG: - PERF: - ]
  RELS: < [ proper_q<0:6> LBL: h4 ARG0: x3 [ x PERS: 3 NUM: sg IND: + ] RSTR: h5␣
↪BODY: h6 ]
          [ named<0:6> LBL: h7 ARG0: x3 CARG: "Abrams" ]
          [ _chase_v_1<7:13> LBL: h1 ARG0: e2 ARG1: x3 ARG2: x9 [ x PERS: 3 NUM: sg␣
↪IND: + ] ]
          [ proper_q<14:20> LBL: h10 ARG0: x9 RSTR: h11 BODY: h12 ]
          [ named<14:20> LBL: h13 ARG0: x9 CARG: "Browne" ] >
  HCONS: < h0 qeq h1 h5 qeq h7 h11 qeq h13 > ]
>>> print(mrx.dumps([x], pretty_print=True))
<mrs-list>
<mrs cfrom="-1" cto="-1"><label vid="0" /><var sort="e" vid="2">
[...]
</mrs>
</mrs-list>
>>> print(dmrx.dumps([x], pretty_print=True))
<dmrs-list>
<dmrs cfrom="-1" cto="-1" index="10002">
[...]
</dmrs>
</dmrs-list>
```

See also:

- Wiki of MRS formats: http://moin.delph-in.net/MrsRfc
- *delphin.mrs.simplemrs* module
- *delphin.mrs.mrx* module
- *delphin.mrs.dmrx* module

Some formats are currently export-only:

```
>>> from delphin.mrs import prolog, simpledmrs
>>> print(prolog.dumps([x], pretty_print=True))
psoa(h0,e2,
```

(continues on next page)

(continued from previous page)

```
  [rel('proper_q',h4,
       [attrval('ARG0',x3),
        attrval('RSTR',h5),
        attrval('BODY',h6)]),
   rel('named',h7,
       [attrval('CARG','Abrams'),
        attrval('ARG0',x3)]),
   rel('_chase_v_1',h1,
       [attrval('ARG0',e2),
        attrval('ARG1',x3),
        attrval('ARG2',x9)]),
   rel('proper_q',h10,
       [attrval('ARG0',x9),
        attrval('RSTR',h11),
        attrval('BODY',h12)]),
   rel('named',h13,
       [attrval('CARG','Browne'),
        attrval('ARG0',x9)])],
  hcons([qeq(h0,h1),qeq(h5,h7),qeq(h11,h13)]))
>>> print(simpledmrs.dumps([x], pretty_print=True))
dmrs {
  [top=10002 index=10002]
  10000 [proper_q<0:6> x PERS=3 NUM=sg IND=+];
  10001 [named<0:6>("Abrams") x PERS=3 NUM=sg IND=+];
  10002 [_chase_v_1<7:13> e SF=prop TENSE=past MOOD=indicative PROG=- PERF=-];
  10003 [proper_q<14:20> x PERS=3 NUM=sg IND=+];
  10004 [named<14:20>("Browne") x PERS=3 NUM=sg IND=+];
  10000:RSTR/H -> 10001;
  10002:ARG1/NEQ -> 10001;
  10002:ARG2/NEQ -> 10004;
  10003:RSTR/H -> 10004;
}
```

See also:

- *delphin.mrs.prolog* module
- *delphin.mrs.simpledmrs* module

PyDelphin also handles basic conversion to Elementary Dependency Structures (EDS). The conversion is lossy, so it's not currently possible to convert from EDS to *MRS. Unlike the export-only formats shown above, however, it is possible to read EDS data and convert to other EDS formats (see below).

```
>>> from delphin.mrs import eds
>>> print(eds.dumps([x], pretty_print=True))
{e2:
 _1:proper_q<0:6>[BV x3]
 x3:named<0:6>("Abrams")[]
 e2:_chase_v_1<7:13>[ARG1 x3, ARG2 x9]
 _2:proper_q<14:20>[BV x9]
 x9:named<14:20>("Browne")[]
}
```

See also:

- *delphin.mrs.eds* module

MRS, DMRS, and EDS all support `to_dict()` and `from_dict()` methods, which make it easy to serialize to JSON.

```
>>> import json
>>> from delphin.mrs import Mrs, Dmrs
>>> print(json.dumps(Mrs.to_dict(x), indent=2))
{
  "relations": [
    {
      "label": "h4",
      "predicate": "proper_q",
[...]
}
>>> print(json.dumps(Dmrs.to_dict(x), indent=2))
{
  "nodes": [
    {
      "nodeid": 10000,
      "predicate": "proper_q",
[...]
}
>>> print(json.dumps(eds.Eds.from_xmrs(x).to_dict(), indent=2))
{
  "top": "e2",
  "nodes": {
    "_1": {
      "label": "proper_q",
      "edges": {
        "BV": "x3"
      },
[...]
}
```

See also:

- *Mrs* class
- *Dmrs* class
- *Eds* class

And finally the dependency representations (DMRS and EDS) have `to_triples()` and `from_triples()` methods, which aid in PENMAN serialization.

```
>>> from delphin.mrs import penman
>>> print(penman.dumps([x], model=Dmrs))
(10002 / _chase_v_1
       :lnk "<7:13>"
       :ARG1-NEQ (10001 / named
                         :lnk "<0:6>"
                         :carg "Abrams"
                         :RSTR-H-of (10000 / proper_q
                                            :lnk "<0:6>"))
       :ARG2-NEQ (10004 / named
                         :lnk "<14:20>"
                         :carg "Browne"
                         :RSTR-H-of (10003 / proper_q
                                            :lnk "<14:20>")))
>>> print(penman.dumps([x], model=eds.Eds))
(e2 / _chase_v_1
    :lnk "<7:13>"
    :ARG1 (x3 / named
```

(continues on next page)

(continued from previous page)

```
                :lnk "<0:6>"
                :carg "Abrams"
                :BV-of (_1 / proper_q
                            :lnk "<0:6>"))
    :ARG2 (x9 / named
                :lnk "<14:20>"
                :carg "Browne"
                :BV-of (_2 / proper_q
                            :lnk "<14:20>")))
```

See also:

- *delphin.mrs.penman* module

2.4 Tokens and Token Lattices

You can inspect the tokens as analyzed by the processor:

```
>>> response.tokens('initial')
<delphin.tokens.YyTokenLattice object at 0x7f3c55abdd30>
>>> print('\n'.join(map(str,response.tokens('initial').tokens)))
(1, 0, 1, <0:6>, 1, "Abrams", 0, "null", "NNP" 1.0000)
(2, 1, 2, <7:13>, 1, "chased", 0, "null", "NNP" 1.0000)
(3, 2, 3, <14:20>, 1, "Browne", 0, "null", "NNP" 1.0000)
```

See also:

- Wiki about YY tokens: http://moin.delph-in.net/PetInput
- *delphin.tokens* module

2.5 Derivations

[incr tsdb()] derivations (unambiguous "recipes" for an analysis with a specific grammar version) are fully modeled:

```
>>> d = response.result(0).derivation()
>>> d.derivation().entity
'sb-hd_mc_c'
>>> d.derivation().daughters
[<UdfNode object (900, hdn_bnp-pn_c, 0.093057, 0, 1) at 139897048235816>, <UdfNode␣
↪object (904, hd-cmp_u_c, -0.846099, 1, 3) at 139897041227960>]
>>> d.derivation().terminals()
[<UdfTerminal object (abrams) at 139897041154360>, <UdfTerminal object (chased) at␣
↪139897041154520>, <UdfTerminal object (browne) at 139897041154680>]
>>> d.derivation().preterminals()
[<UdfNode object (71, abrams, 0.0, 0, 1) at 139897041214040>, <UdfNode object (52,␣
↪chase_v1, 0.0, 1, 2) at 139897041214376>, <UdfNode object (70, browne, 0.0, 2, 3)␣
↪at 139897041214712>]
```

See also:

- Wiki about derivations: http://moin.delph-in.net/ItsdbDerivations
- *delphin.derivation* module

2.6 [incr tsdb()] TestSuites

PyDelphin has full support for reading and writing [incr tsdb()] testsuites:

```
>>> from delphin import itsdb
>>> ts = itsdb.TestSuite('erg/tsdb/gold/mrs')
>>> len(ts['item'])
107
>>> ts['item'][0]['i-input']
'It rained.'
>>> ts.write(tables=itsdb.tsdb_core_files, path='mrs-skeleton')
```

See also:

- [incr tsdb()] wiki: http://moin.delph-in.net/ItsdbTop
- `delphin.itsdb` module
- *Working with [incr tsdb()] Testsuites* tutorial

2.7 TSQL Queries

Partial support of the Test Suite Query Language (TSQL) allows for easy selection of [incr tsdb()] TestSuite data.

```
>>> from delphin import itsdb, tsql
>>> ts = itsdb.TestSuite('erg/tsdb/gold/mrs')
>>> next(tsql.select('i-id i-input where i-length > 5 && readings > 0', ts))
[61, 'Abrams handed the cigarette to Browne.']
```

See also:

- TSQL documentation: http://www.delph-in.net/tsnlp/ftp/manual/volume2.ps.gz
- `delphin.tsql` module

2.8 Regular Expression Preprocessors (REPP)

PyDelphin provides a full implementation of Regular Expression Preprocessors (REPP), including correct characterization and the loading from PET configuration files. Unique to PyDelphin (I think) is the ability to trace through an application of the tokenization rules.

```
>>> from delphin import repp
>>> r = repp.REPP.from_config('../../grammars/erg/pet/repp.set')
>>> for tok in r.tokenize("Abrams didn't chase Browne.").tokens:
...     print(tok.form, tok.lnk)
...
Abrams <0:6>
did <7:10>
n’t <10:13>
chase <14:19>
Browne <20:26>
. <26:27>
>>> for step in r.trace("Abrams didn't chase Browne."):
...     if isinstance(step, repp.REPPStep):
```

(continues on next page)

(continued from previous page)

```
...         print('{}\t-> {}\t{}'.format(step.input, step.output, step.operation))
...
Abrams didn't chase Browne.     ->  Abrams didn't chase Browne.         !^(.+)$
↪   \1
 Abrams didn't chase Browne.    ->  Abrams didn’t chase Browne.         !'
↪  ’
 Abrams didn't chase Browne.    ->  Abrams didn’t chase Browne.         Internal␣
↪group #1
 Abrams didn't chase Browne.    ->  Abrams didn’t chase Browne.         Internal␣
↪group #1
 Abrams didn't chase Browne.    ->  Abrams didn’t chase Browne.         Module quotes
 Abrams didn’t chase Browne.    ->   Abrams didn’t chase Browne.        !^(.+)$
↪   \1
  Abrams didn’t chase Browne.   ->  Abrams didn’t chase Browne.         !  +
 Abrams didn’t chase Browne.    ->  Abrams didn’t chase Browne .        !([^ ])(\.)␣
↪([])}”"’'... ]*)$              \1 \2 \3
 Abrams didn’t chase Browne.    ->  Abrams didn’t chase Browne .        Internal␣
↪group #1
 Abrams didn’t chase Browne.    ->  Abrams didn’t chase Browne .        Internal␣
↪group #1
 Abrams didn’t chase Browne .   ->  Abrams did n’t chase Browne .       !([^␣
↪])([nN])[’’]([tT])                 \1 \2’\3
Abrams didn't chase Browne.     ->  Abrams did n’t chase Browne .       Module␣
↪tokenizer
```

Note that the trace shows the sequential order of rule applications, but not the tree-like branching of REPP modules.

See also:

- REPP wiki: http://moin.delph-in.net/ReppTop
- Wiki for PET’s REPP configuration: http://moin.delph-in.net/ReppPet
- *delphin.repp* module

2.9 Type Description Language (TDL)

The TDL language is fairly simple, but the interpretation of type hierarchies (feature inheritance, re-entrancies, unification and subsumption) can be very complex. PyDelphin has partial support for reading TDL files. It can read nearly any kind of TDL in a DELPH-IN grammar (type definitions, lexicons, transfer rules, etc.), but it does not do any interpretation. It can be useful for static code analysis.

```
>>> from delphin import tdl
>>> lex = {}
>>> for event, obj, lineno in tdl.iterparse('erg/lexicon.tdl'):
...     if event == 'TypeDefinition':
...         lex[obj.identifier] = obj
...
>>> len(lex)
40234
>>> lex['cactus_n1']
<TypeDefinition object 'cactus_n1' at 140226925196400>
>>> lex['cactus_n1'].supertypes
[<TypeIdentifier object (n_-_c_le) at 140226925284232>]
>>> lex['cactus_n1'].features()
[('ORTH', <ConsList object at 140226925534472>), ('SYNSEM', <AVM object at␣
↪140226925299464>)]
```

(continues on next page)

(continued from previous page)

```
>>> lex['cactus_n1']['ORTH'].features()
[('FIRST', <String object (cactus) at 140226925284352>), ('REST', None)]
>>> lex['cactus_n1']['ORTH'].values()
[<String object (cactus) at 140226925284352>]
>>> lex['cactus_n1']['ORTH.FIRST']
<String object (cactus) at 140226925284352>
>>> print(tdl.format(lex['cactus_n1']))
cactus_n1 := n_-_c_le &
  [ ORTH < "cactus" >,
    SYNSEM [ LKEYS.KEYREL.PRED "_cactus_n_1_rel",
             LOCAL.AGR.PNG png-irreg,
             PHON.ONSET con ] ].
```

See also:

- A semi-formal specification of TDL: http://moin.delph-in.net/TdlRfc
- A grammar-engineering FAQ about TDL: http://moin.delph-in.net/GeFaqTdlSyntax
- *delphin.tdl* module

2.10 Semantic Interfaces (SEM-I)

A grammar's semantic model is encoded in the predicate inventory and constraints of the grammar, but as the interpretation of a grammar is non-trivial (see *Type Description Language (TDL)* above), using the grammar to validate semantic representations is a significant burden. A semantic interface (SEM-I) is a distilled and simplified representation of a grammar's semantic model, and is thus a useful way to ensure that grammar-external semantic representations are valid with respect to the grammar. PyDelphin supports the reading and inspection of SEM-Is.

```
>>> from delphin.mrs import semi
>>> s = semi.load('../../grammars/erg/etc/erg.smi')
>>> list(s.variables)
['u', 'i', 'p', 'h', 'e', 'x']
>>> list(s.roles)
['ARG0', 'ARG1', 'ARG2', 'ARG3', 'ARG4', 'ARG', 'BODY', 'CARG', 'L-HNDL', 'L-INDEX',
↪'R-HNDL', 'R-INDEX', 'RSTR']
>>> s.roles['ARG3']
Role(rargname='ARG3', value='u', proplist=[], optional=False)
>>> list(s.properties)
['bool', '+', '-', 'tense', 'tensed', 'past', 'pres', 'fut', 'untensed', 'mood',
↪'subjunctive', 'indicative', 'gender', 'm-or-f', 'm', 'f', 'n', 'number', 'sg', 'pl
↪', 'person', '1', '2', '3', 'pt', 'refl', 'std', 'zero', 'sf', 'prop-or-ques', 'prop
↪', 'ques', 'comm']
>>> s.properties['fut']
Property(type='fut', supertypes=('tensed',))
>>> len(s.predicates)
22539
>>> s.predicates['_cactus_n_1']
Predicate(predicate='_cactus_n_1', supertypes=(), synopses=[(Role(rargname='ARG0',␣
↪value='x', proplist=[('IND', '+')], optional=False),)])
```

See also:

- The SEM-I wiki: http://moin.delph-in.net/SemiRfc
- *delphin.mrs.semi* module

CHAPTER 3

Using ACE from PyDelphin

ACE is one of the most efficient processors for DELPH-IN grammars, and has an impressively fast start-up time. PyDelphin tries to make it easier to use ACE from Python with the *delphin.interfaces.ace* module, which provides functions and classes for compiling grammars, parsing, transfer, and generation.

In this tutorial, `delphin.interfaces.ace` is assumed to be imported as `ace`, as in the following:

```
>>> from delphin.interfaces import ace
```

3.1 Compiling a Grammar

The *compile()* function can be used to compile a grammar from its source. It takes two arguments, the location of the ACE configuration file and the path of the compiled grammar to be written. For instance (assume the current working directory is the grammar directory):

```
>>> ace.compile('ace/config.tdl', 'zhs.dat')
```

This is equivalent to running the following from the commandline (again, from the grammar directory):

```
[~/zhong/cmn/zhs/]$ ace -g ace/config.tdl -G zhs.dat
```

All of the following topics assume that a compiled grammar exists.

3.2 Parsing

The ACE interface handles the interaction between Python and ACE, giving ACE the arguments to parse and then interpreting the output back into Python data structures.

The easiest way to parse a single sentence is with the *parse()* function. Its first argument is the path to the compiled grammar, and the second is the string to parse:

```
>>> response = ace.parse('zhs.dat', '  ')
>>> len(response['results'])
8
>>> response['results'][0]['mrs']
'[ LTOP: h0 INDEX: e2 [ e SF: prop-or-ques E.ASPECT: perfective ] RELS: < [ "__n_1_rel
↪"<0:1> LBL: h4 ARG0: x3 [ x SPECI: + SF: prop COG-ST: uniq-or-more PNG.PERNUM:␣
↪pernum PNG.GENDER: gender PNG.ANIMACY: animacy ] ]  [ generic_q_rel<-1:-1> LBL: h5␣
↪ARG0: x3 RSTR: h6 BODY: h7 ]  [ "__v_3_rel"<2:3> LBL: h1 ARG0: e2 ARG1: x3 ARG2: x8␣
↪[ x SPECI: bool SF: prop COG-ST: cog-st PNG.PERNUM: pernum PNG.GENDER: gender PNG.
↪ANIMACY: animacy ] ] > HCONS: < h0 qeq h1 h6 qeq h4 > ICONS: < e2 non-focus x8 > ]'
```

Notice that the response is a Python dictionary. They are in fact a subclass of dictionaries with some added convenience methods. Using dictionary access methods returns the raw data, but the function access can simplify interpretation of the results. For example:

```
>>> len(response.results())
8
>>> response.result(0).mrs()
<Mrs object ( generic ) at 2567183400998>
```

These response objects are described in the documentation for the *interfaces* package.

In addition to single sentences, a sequence of sentences can be parsed, yielding a sequence of results, using *parse_from_iterable()*:

```
>>> for response in ace.parse_from_iterable('zhs.dat', ['  ', ' ']):
...     print(len(response.results()))
...
8
5
```

Both *parse()* and *parse_from_iterable()* use the *AceParser* class for interacting with ACE. This class can also be instantiated directly and interacted with as long as the process is open, but don't forget to close the process when done.

```
>>> parser = ace.AceParser('zhs.dat')
>>> len(parser.interact('  ').results())
8
>>> parser.close()
0
```

The class can also be used as a context manager, which removes the need to explicitly close the ACE process.

```
>>> with ace.AceParser('zhs.dat') as parser:
...     print(len(parser.interact('  ').results()))
...
8
```

The *AceParser* class and *parse()* and *parse_from_iterable()* functions all take additional arguments for affecting how ACE is accessed, e.g., for selecting the location of the ACE binary, setting command-line options, and changing the environment variables of the subprocess:

```
>>> with ace.AceParser('zhs-0.9.26.dat',
...                    executable='/opt/ace-0.9.26/ace',
...                    cmdargs=['-n', '3', '--timeout', '5']) as parser:
...     print(len(parser.interact('  ').results()))
```

(continues on next page)

(continued from previous page)

```
...
5
```

See the `delphin.interfaces.ace` module documentation for more information about options for `AceParser`.

3.3 Generation

Generating sentences from semantics is similar to parsing, but the `simplemrs` serialization of the semantics is given as input instead of sentences. You can generate from a single semantic representation with `generate()`:

```
>>> m = '''
... [ LTOP: h0
...   RELS: < [ "_rain_v_1_rel" LBL: h1 ARG0: e2 [ e TENSE: pres ] ] >
...   HCONS: < h0 qeq h1 > ]'''
>>> response = ace.generate('erg.dat', m)
>>> response.result(0)['surface']
'It rains.'
```

The response object is the same as with parsing. You can also generate from a list of MRSs with `generate_from_iterable()`:

```
>>> responses = list(ace.generate_from_iterable('erg.dat', [m, m]))
>>> len(responses)
2
```

Or instantiate a generation process with `AceGenerator`:

```
>>> with ace.AceGenerator('erg.dat') as generator:
...     print(generator.iteract(m).result(0)['surface'])
...
It rains.
```

3.4 Transfer

ACE also implements most of the LOGON transfer formalism, and this functionality is available in PyDelphin via the `AceTransferer` class and related functions. In the current version of ACE, transfer does not return as much information as with parsing and generation, but the response object in PyDelphin is the same as with the other tasks.

```
>>> j_response = ace.parse('jacy.dat', '  ')
>>> je_response = ace.transfer('jaen.dat', j_response.result(0)['mrs'])
>>> e_response = ace.generate('erg.dat', je_response.result(0)['mrs'])
>>> e_response.result(0)['surface']
'It rains.'
```

3.5 Tips and Tricks

Sometimes the input data needs to be modified before it can be parsed, such as the morphological segmentation of Japanese text. Users may also wish to modify the results of processing, such as to streamline an MRS–DMRS conversion pipeline. The former is an example of a preprocessor and the latter a postprocessor. There can also be

"coprocessors" that execute alongside the original, such as for returning the result of a statistical parser when the original fails to reach a parse. It is straightforward to accomplish all of these configurations with Python and PyDelphin, but the resulting pipeline may not be compatible with other interfaces, such as *TestSuite.process()*. By using the `delphin.interfaces.base.Process` class to wrap an *AceProcess* instance, these pre-, co-, and post-processors can be implemented in a more useful way. See *Wrapping a Processor for Preprocessing* for an example of using `Process` as a preprocessor.

3.6 Troubleshooting

Some environments have an encoding that isn't compatible with what ACE expects. One way to mitigate this issue is to pass in the appropriate environment variables via the `env` parameter. For example:

```
>>> import os
>>> env = os.environ
>>> env['LANG'] = 'en_US.UTF8'
>>> ace.parse('zhs.dat', '  ', env=env)
```

CHAPTER 4

PyDelphin at the Command Line

While PyDelphin primarily exists as a library to be used by other software, for convenience it also provides a top-level script for some common actions. There are two ways to call this script, depending on how you installed PyDelphin:

- `delphin.sh` - if you've downloaded the PyDelphin source files, this script is available in the top directory of the code
- `delphin` - if you've installed PyDelphin via `pip`, it is available system-wide as `delphin` (without the `.sh`)

For this tutorial, it is assumed you've installed PyDelphin via `pip` and thus use the second form.

The `delphin` command has several subcommands, described below.

See also:

The *`delphin.commands`* module provides a Python API for these commands.

4.1 convert

The `convert` subcommand enables conversion of various *MRS representations. You provide `-from` and `-to` arguments to select the representations (the default for both is `simplemrs`). Here is an example of converting SimpleMRS to JSON-serialized DMRS:

```
$ echo '[ "It rains." TOP: h0 RELS: < [ _rain_v_1<3:8> LBL: h1 ARG0: e2 ] > HCONS: <␣
↪h0 qeq h1 > ]' \
  | delphin convert --to dmrs-json
[{"surface": "It rains.", "links": [{"to": 10000, "rargname": null, "from": 0, "post
↪": "H"}], "nodes": [{"sortinfo": {"cvarsort": "e"}, "lnk": {"to": 8, "from": 3},
↪"nodeid": 10000, "predicate": "_rain_v_1"}]}]
```

As the default for `-from` and `-to` is `simplemrs`, it can be used to easily "pretty-print" an MRS (if you execute this in a terminal, you'll notice syntax highlighting as well):

```
$ echo '[ "It rains." TOP: h0 RELS: < [ _rain_v_1<3:8> LBL: h1 ARG0: e2 ] > HCONS: <␣
↪h0 qeq h1 > ]' \
  | delphin convert --pretty-print
[ "It rains."
  TOP: h0
  RELS: < [ _rain_v_1<3:8> LBL: h1 ARG0: e2 ] >
  HCONS: < h0 qeq h1 > ]
```

Some formats are export-only, such as `dmrs-tikz`:

```
$ echo '[ "It rains." TOP: h0 RELS: < [ _rain_v_1<3:8> LBL: h1 ARG0: e2 ] > HCONS: <␣
↪h0 qeq h1 > ]' \
  | delphin convert --to dmrs-tikz
\documentclass{standalone}

\usepackage{tikz-dependency}
\usepackage{relsize}
[...]
```

See `delphin convert -help` for more information.

4.2 select

The `select` subcommand selects data from an [incr tsdb()] profile using TSQL queries. For example, if you want to get the `i-id` and `i-input` fields from a profile, do this:

```
$ delphin select 'i-id i-input from item' ~/grammars/jacy/tsdb/gold/mrs/
11@
21@
[..]
```

In many cases, the `from` clause of the query is not necessary, and the appropriate tables will be selected automatically. Fields from multiple tables can be used and the tables containing them will be automatically *joined*:

```
$ delphin select 'i-id mrs' ~/grammars/jacy/tsdb/gold/mrs/
11@[ LTOP: h1 INDEX: e2 ... ]
[..]
```

The results can be filtered by providing `where` clauses:

```
$ delphin select 'i-id i-input where i-input ~ ""' ~/grammars/jacy/tsdb/gold/mrs/
11@
71@
81@
```

See `delphin select -help` for more information.

4.3 mkprof

Rather than selecting data to send to stdout, you can also output a new [incr tsdb()] profile with the `mkprof` subcommand. If a profile is given via the `-source` option, the relations file of the source profile is used by default, and you may use a `--where` option to use TSQL conditions to filter the data used in creating the new profile. Otherwise, the `-relations` option is required, and the input may be a file of sentences via the `-input` option, or a stream

of sentences via stdin. Sentences via file or stdin can be prefixed with an asterisk, in which case they are considered ungrammatical (`i-wf` is set to `0`). Here is an example:

```
$ echo -e "A dog barks.\n*Dog barks a." \
  | delphin mkprof \
      --relations ~/logon/lingo/lkb/src/tsdb/skeletons/english/Relations \
      newprof
9746   bytes  relations
67     bytes  item
```

Using `--where`, sub-profiles can be created, which may be useful for testing different parameters. For example, to create a sub-profile with only items of less than 10 words, do this:

```
$ delphin mkprof --where 'i-length < 10' \
                 --source ~/grammars/jacy/tsdb/gold/mrs/ \
                 mrs-short
9067   bytes  relations
12515  bytes  item
```

See `delphin mkprof -help` for more information.

4.4 process

PyDelphin can use ACE to process [incr tsdb()] testsuites. As with the art utility, the workflow is to first create an empty testsuite (see *mkprof* above), then to process that testsuite in place.

```
$ delphin mkprof -s erg/tsdb/gold/mrs/ mrs-parsed
 9746  bytes  relations
 10810 bytes  item
 [...]
$ delphin process -g erg-1214-x86-64-0-9.27.dat mrs-parsed
NOTE: parsed 107 / 107 sentences, avg 3253k, time 2.50870s
```

The default task is parsing, but transfer and generation are also possible. For these, it is suggested to create a separate output testsuite for the results, as otherwise it would overwrite the `results` table. Generation is activated with the `-e` option, and the `-s` option selects the source profile.

```
$ delphin mkprof -s erg/tsdb/gold/mrs/ mrs-generated
 9746  bytes  relations
 10810 bytes  item
 [...]
$ delphin process -g erg-1214-x86-64-0-9.27.dat -e -s mrs-parsed mrs-generated
NOTE: 77 passive, 132 active edges in final generation chart; built 77 passives total.
↪ [1 results]
NOTE: 59 passive, 139 active edges in final generation chart; built 59 passives total.
↪ [1 results]
[...]
NOTE: generated 440 / 445 sentences, avg 4880k, time 17.23859s
NOTE: transfer did 212661 successful unifies and 244409 failed ones
```

See `delphin process -help` for more information.

See also:

The art utility and [incr tsdb()] are other testsuite processors with different kinds of functionality.

4.5 compare

The `compare` subcommand is a lightweight way to compare bags of MRSs, e.g., to detect changes in a profile run with different versions of the grammar.

```
$ delphin compare ~/grammars/jacy/tsdb/current/mrs/ \
                  ~/grammars/jacy/tsdb/gold/mrs/
11  <1,0,1>
21  <1,0,1>
31  <3,0,1>
[..]
```

See `delphin compare -help` for more information.

See also:

The gTest application is a more fully-featured profile comparer, as is [incr tsdb()] itself.

4.6 repp

A regular expression preprocessor (REPP) can be used to tokenize input strings.

```
$ delphin repp -c erg/pet/repp.set --format triple <<< "Abrams didn't chase Browne."
(0, 6, Abrams)
(7, 10, did)
(10, 13, n’t)
(14, 19, chase)
(20, 26, Browne)
(26, 27, .)
```

PyDelphin is not as fast as the C++ implementation, but its tracing functionality can be useful for debugging.

```
$ delphin repp -c erg/pet/repp.set --trace <<< "Abrams didn't chase Browne."
Applied:!^(.+)$                \1
   In:Abrams didn't chase Browne.
  Out: Abrams didn't chase Browne.
Applied:!'            ’
   In: Abrams didn't chase Browne.
  Out: Abrams didn’t chase Browne.
Applied:Internal group #1
   In: Abrams didn't chase Browne.
  Out: Abrams didn’t chase Browne.
Applied:Internal group #1
   In: Abrams didn't chase Browne.
  Out: Abrams didn’t chase Browne.
Applied:Module quotes
   In: Abrams didn't chase Browne.
  Out: Abrams didn’t chase Browne.
Applied:!^(.+)$                \1
   In: Abrams didn’t chase Browne.
  Out:  Abrams didn’t chase Browne.
Applied:!  +
   In:  Abrams didn’t chase Browne.
  Out: Abrams didn’t chase Browne.
Applied:!([^ ])(\.) ([])}”"’'... ]*)$           \1 \2 \3
```

(continues on next page)

(continued from previous page)

```
   In: Abrams didn’t chase Browne.
  Out: Abrams didn’t chase Browne .
Applied:Internal group #1
   In: Abrams didn’t chase Browne.
  Out: Abrams didn’t chase Browne .
Applied:Internal group #1
   In: Abrams didn’t chase Browne.
  Out: Abrams didn’t chase Browne .
Applied:!([^ ])([nN])[’']([tT])              \1 \2’\3
   In: Abrams didn’t chase Browne .
  Out: Abrams did n’t chase Browne .
Applied:Module tokenizer
   In:Abrams didn't chase Browne.
  Out: Abrams did n’t chase Browne .
Done: Abrams did n’t chase Browne .
```

See `delphin repp -help` for more information.

See also:

- The C++ REPP implementation: http://moin.delph-in.net/ReppTop#REPP_in_PET_and_Stand-Alone

CHAPTER 5

Working with [incr tsdb()] Testsuites

[incr tsdb()] is the canonical software for managing **testsuites**—collections of test items for judging the performance of an implemented grammar—within DELPH-IN. While the original purpose of testsuites is to aid in grammar development, they are also useful more generally for batch processing. PyDelphin has good support for a range of [incr tsdb()] functionality.

There are several words in use to describe testsuites:

skeleton a testsuite containing only input items and static annotations, such as for indicating grammaticality or listing exemplified phenomena, ready to be processed

profile a testsuite augmented with analyses from a grammar; useful for inspecting the grammar's competence and performance, or for building treebanks

testsuite a general term for both skeletons and profiles

For this tutorial, it is assumed you've imported the `delphin.itsdb` module as follows:

```
>>> from delphin import itsdb
```

See also:

- The [incr tsdb()] homepage: http://www.delph-in.net/itsdb/
- The [incr tsdb()] wiki: http://moin.delph-in.net/ItsdbTop

5.1 Loading and Inspecting Testsuites

Loading a testsuite is as simple as creating a `TestSuite` object with the directory as its argument:

```
>>> ts = itsdb.TestSuite('erg/tsdb/gold/mrs')
```

The `TestSuite` object loads both the relations file (i.e., the database schema) and the data tables themselves. The relations can be inspected via the `TestSuite.relations` attribute:

```
>>> ts.relations.tables
('item', 'analysis', 'phenomenon', 'parameter', 'set', 'item-phenomenon', 'item-set',
↪'run', 'parse', 'result', 'rule', 'output', 'edge', 'tree', 'decision', 'preference
↪', 'update', 'fold', 'score')
>>> [field.name for field in ts.relations['phenomenon']]
['p-id', 'p-name', 'p-supertypes', 'p-presupposition', 'p-interaction', 'p-purpose',
↪'p-restrictions', 'p-comment', 'p-author', 'p-date']
```

Key lookups on the testsuite return the *Table* whose name is the given key. Note that the table name is as it appears in the relations file and not the table's filename (e.g., in case the table is compressed and the filename has a `.gz` extension).

```
>>> len(ts['item'])
107
>>> ts['item'][0]['i-input']
'It rained.'
```

The *select()* method allows for slightly more flexible queries, such as those involving more than one field. The method returns a Python generator for efficient iteration, so in the following example it needs to be cast to a list for slicing.

```
>>> list(ts.select('item:i-id@i-input'))[0:3]
[[11, 'It rained.'], [21, 'Abrams barked.'], [31, 'The window opened.']]
```

Sometimes the desired fields exist in different tables, such as when one wants to pair an input item identifier with its results—a one-to-many mapping. In these cases, the *delphin.itsdb.join()* function is useful. It returns a new *Table* with the joined data, and the field names are prefixed with their source table names.

```
>>> joined = itsdb.join(ts['parse'], ts['result'])
>>> next(joined.select('parse:i-id@result:mrs'))
[11, '[ TOP: h1 INDEX: e3 [ e SF: PROP TENSE: PAST MOOD: INDICATIVE PROG: - PERF: - ]␣
↪RELS: < [ _rain_v_1<3:10> LBL: h2 ARG0: e3 ] > HCONS: < h1 qeq h2 > ]']
```

See also:

The *delphin.tsql* module provides support for TSQL queries over test suites which are much more flexible and powerful than the *TestSuite.select* method.

5.2 Writing Testsuites to Disk

See also:

The *mkprof* command is a more versatile method of creating testsuites at the command line.

The *write()* method of TestSuites is the primary way of writing in-memory TestSuite data to disk. Its most basic form writes all tables to the path used to load the testsuite:

```
>>> from delphin import itsdb
>>> ts = itsdb.TestSuite('tsdb/gold/mrs')
>>> ts.write()
```

This method does not work if the testsuite was created entirely in-memory (i.e., if it has no `path`). In this case, or also in the case where a different destination is desired, the `path` can be specified as a parameter:

```
>>> ts.write(path='tsdb/current/mrs')
```

The first parameter to the `write()` method is a description of what to write. It could be a single table name, a list of table names, or a mapping from table names to lists of table records:

```
>>> ts.write('item')  # only write the 'item' file
>>> ts.write(['item', 'parse'])  # only write 'item' and 'parse'
>>> ts.write({'result': result_records})  # write result_records to 'result'
```

By default, writing a table deletes any previous contents, so the entire file contents need to be written at once. If you want to write results one-by-one, the `append` parameter is useful. You many need to clear the in-memory table before appending the first time, and this can be done by writing an empty list with `append=False`:

```
>>> ts.write({'result': [], append=False)  # to erase on-disk table
>>> ts.['result'][:] = []                  # to clear in-memory table
>>> for record in result_records:
...     ts.write({'result': [record]}, append=True)
```

5.3 Processing Testsuites with ACE

PyDelphin has the ability to process testsuites using ACE, similar to the art utility and [incr tsdb()] itself. The simplest method is to pass in a running *`AceProcess`* instance to *`TestSuite.process`*—the testsuite class will determine if the processor is for parsing, transfer, or generation (using the `AceProcessor.task` attribute) and select the appropriate inputs from the testsuite.

```
>>> from delphin import itsdb
>>> from delphin.interfaces import ace
>>> ts = itsdb.TestSuite('tsdb/skeletons/matrix')
>>> with ace.AceParser('indra.dat') as cpu:
...     ts.process(cpu)
...
NOTE: parsed 2 / 3 sentences, avg 887k, time 0.04736s
>>> ts.write(path='tsdb/current/matrix')
```

Processing a testsuite that has a path (that is, backed by files on disk) will write the results to disk. Processing an in-memory testsuite will store the results in-memory. For other options please see the API documentation for *`TestSuite.process`*, specifically the `buffer_size` parameter. When the results are all in-memory, you can write them to disk with TestSuite's *`write()`* method with the `path` parameter.

> **Warning:** PyDelphin does not prevent or warn you about overwriting skeletons or gold profiles, so take care when using the `write()` method without the `path` parameter.

If you have a testsuite object `ts` and call `ts.process()`, both the source items and the results are stored in `ts`. For parsing this isn't a problem because the source items and results are located in different tables, but for transfering or generating you may want to use the `source` parameter in order to select inputs from a separate testsuite than the one where results will be stored:

```
>>> from delphin.interfaces import ace
>>> from delphin import itsdb
>>> src_ts = itsdb.TestSuite('tsdb/current/mrs')
>>> tgt_ts = itsdb.TestSuite('tsdb/current/mrs-gen')
>>> with ace.AceGenerator('jacy-0.9.27.dat') as cpu:
...     tgt_ts.process(cpu, source=src_ts)
...
```

(continues on next page)

(continued from previous page)

```
NOTE: 75 passive, 361 active edges in final generation chart; built 89 passives total.
↪ [1 results]
NOTE: 35 passive, 210 active edges in final generation chart; built 37 passives total.
↪ [1 results]
[...]
```

CHAPTER 6

delphin.commands

PyDelphin API counterparts to the `delphin` commands.

The public functions in this module largely mirror the front-end subcommands provided by the `delphin` command, with some small changes to argument names or values to be better-suited to being called from within Python.

`delphin.commands.`**`compare`**(*testsuite*, *gold*, *select='i-id i-input mrs'*)

Compare two [incr tsdb()] profiles.

Parameters

- **testsuite** (*str, TestSuite*) – path to the test [incr tsdb()] testsuite or a `TestSuite` object
- **gold** (*str, TestSuite*) – path to the gold [incr tsdb()] testsuite or a `TestSuite` object
- **select** – TSQL query to select (id, input, mrs) triples (default: `i-id i-input mrs`)

Yields *dict* –

Comparison results as:

```
{"id": "item identifier",
 "input": "input sentence",
 "test": number_of_unique_results_in_test,
 "shared": number_of_shared_results,
 "gold": number_of_unique_results_in_gold}
```

`delphin.commands.`**`convert`**(*path*, *source_fmt*, *target_fmt*, *select='result:mrs'*, *properties=True*, *show_status=False*, *predicate_modifiers=False*, *color=False*, *pretty_print=False*, *indent=None*)

Convert between various DELPH-IN Semantics representations.

Parameters

- **path** (*str, file*) – filename, testsuite directory, open file, or stream of input representations

- **source_fmt** (*str*) – convert from this format
- **target_fmt** (*str*) – convert to this format
- **select** (*str*) – TSQL query for selecting data (ignored if *path* is not a testsuite directory; default: "result:mrs")
- **properties** (*bool*) – include morphosemantic properties if True (default: True)
- **show_status** (*bool*) – show disconnected EDS nodes (ignored if *target_fmt* is not "eds"; default: False)
- **predicate_modifiers** (*bool*) – apply EDS predicate modification for certain kinds of patterns (ignored if *target_fmt* is not an EDS format; default: False)
- **color** (*bool*) – apply syntax highlighting if True and *target_fmt* is "simplemrs" (default: False)
- **pretty_print** (*bool*) – if True, format the output with newlines and default indentation (default: False)
- **indent** (*int, optional*) – specifies an explicit number of spaces for indentation (implies *pretty_print*)

Returns *str* – the converted representation

delphin.commands.**mkprof**(*destination*, *source=None*, *relations=None*, *where=None*, *in_place=False*, *skeleton=False*, *full=False*, *gzip=False*)

Create [incr tsdb()] profiles or skeletons.

Data for the testsuite may come from an existing testsuite or from a list of sentences. There are four main usage patterns:

- source="testsuite/" – read data from testsuite/
- source=None, in_place=True – read data from *destination*
- source=None, in_place=False – read sentences from stdin
- source="sents.txt" – read sentences from sents.txt

For the latter two, the *relations* parameter must be specified.

Parameters

- **destination** (*str*) – path of the new testsuite
- **source** (*str*) – path to a source testsuite or a file containing sentences; if not given and *in_place* is False, sentences are read from stdin
- **relations** (*str*) – path to a relations file to use for the created testsuite; if None and *source* is given, the relations file of the source testsuite is used
- **where** (*str*) – TSQL condition to filter records by; ignored if *source* is not a testsuite
- **in_place** (*bool*) – if True and *source* is not given, use *destination* as the source for data (default: False)
- **skeleton** (*bool*) – if True, only write tsdb-core files (default: False)
- **full** (*bool*) – if True, copy all data from the source testsuite (requires *source* to be a testsuite path; default: False)
- **gzip** (*bool*) – if True, non-empty tables will be compressed with gzip

`delphin.commands.`**`process`**(*grammar*, *testsuite*, *source=None*, *select=None*, *generate=False*, *transfer=False*, *options=None*, *all_items=False*, *result_id=None*, *gzip=False*)

Process (e.g., parse) a [incr tsdb()] profile.

Results are written to directly to *testsuite*.

If *select* is `None`, the defaults depend on the task:

Task	Default value of *select*
Parsing	`item:i-input`
Transfer	`result:mrs`
Generation	`result:mrs`

Parameters

- **`grammar`** (*`str`*) – path to a compiled grammar image
- **`testsuite`** (*`str`*) – path to a [incr tsdb()] testsuite where data will be read from (see *source*) and written to
- **`source`** (*`str`*) – path to a [incr tsdb()] testsuite; if `None`, *testsuite* is used as the source of data
- **`select`** (*`str`*) – TSQL query for selecting processor inputs (default depends on the processor type)
- **`generate`** (*`bool`*) – if `True`, generate instead of parse (default: `False`)
- **`transfer`** (*`bool`*) – if `True`, transfer instead of parse (default: `False`)
- **`options`** (*`list`*) – list of ACE command-line options to use when invoking the ACE subprocess; unsupported options will give an error message
- **`all_items`** (*`bool`*) – if `True`, don't exclude ignored items (those with `i-wf==2`) when parsing
- **`result_id`** (*`int`*) – if given, only keep items with the specified `result-id`
- **`gzip`** (*`bool`*) – if `True`, non-empty tables will be compressed with gzip

`delphin.commands.`**`repp`**(*file*, *config=None*, *module=None*, *active=None*, *format=None*, *trace_level=0*)

Tokenize with a Regular Expression PreProcessor (REPP).

Results are printed directly to stdout. If more programmatic access is desired, the *`delphin.repp`* module provides a similar interface.

Parameters

- **`file`** (*`str, file`*) – filename, open file, or stream of sentence inputs
- **`config`** (*`str`*) – path to a PET REPP configuration (.set) file
- **`module`** (*`str`*) – path to a top-level REPP module; other modules are found by external group calls
- **`active`** (*`list`*) – select which modules are active; if `None`, all are used; incompatible with *config* (default: `None`)
- **`format`** (*`str`*) – the output format (`"yy"`, `"string"`, `"line"`, or `"triple"`; default: `"yy"`)
- **`trace_level`** (*`int`*) – if `0` no trace info is printed; if `1`, applied rules are printed, if greather than `1`, both applied and unapplied rules (in order) are printed (default: `0`)

`delphin.commands.`**`select`**(*dataspec*, *testsuite*, *mode='list'*, *cast=True*)

Select data from [incr tsdb()] profiles.

Parameters

- **query** (*str*) – TSQL select query (e.g., `'i-id i-input mrs'` or `'* from item where readings > 0'`)
- **testsuite** (*str, TestSuite*) – testsuite or path to testsuite containing data to select
- **mode** (*str*) – see *delphin.itsdb.select_rows()* for a description of the *mode* parameter (default: `list`)
- **cast** (*bool*) – if `True`, cast column values to their datatype according to the relations file (default: `True`)

Returns a generator that yields selected data

CHAPTER 7

delphin.derivation

Classes and functions related to derivation trees.

Derivation trees represent a unique analysis of an input using an implemented grammar. They are a kind of syntax tree, but as they use the actual grammar entities (e.g., rules or lexical entries) as node labels, they are more specific than trees using general category labels (e.g., "N" or "VP"). As such, they are more likely to change across grammar versions.

See also:

More information about derivation trees is found at http://moin.delph-in.net/ItsdbDerivations

For the following Japanese example…

```
tooku   ni  juusei  ga  kikoe-ta
distant LOC gunshot NOM can.hear-PFV
"Shots were heard in the distance."
```

… here is the derivation tree of a parse from Jacy in the Unified Derivation Format (UDF):

```
(utterance-root
 (564 utterance_rule-decl-finite 1.02132 0 6
  (563 hf-adj-i-rule 1.04014 0 6
   (557 hf-complement-rule -0.27164 0 2
    (556 quantify-n-rule 0.311511 0 1
     (23 tooku_1 0.152496 0 1
      ("" 0 1)))
    (42 ni-narg 0.478407 1 2
     ("" 1 2)))
   (562 head_subj_rule 1.512 2 6
    (559 hf-complement-rule -0.378462 2 4
     (558 quantify-n-rule 0.159015 2 3
      (55 juusei_1 0 2 3
       ("" 2 3)))
     (56 ga 0.462257 3 4
```

(continues on next page)

(continued from previous page)

```
      ("" 3 4)))
    (561 vstem-vend-rule 1.34202 4 6
     (560 i-lexeme-v-stem-infl-rule 0.365568 4 5
      (65 kikoeru-stem 0 4 5
       ("" 4 5)))
     (81 ta-end 0.0227589 5 6
      ("" 5 6)))))))
```

In addition to the UDF format, there is also the UDF export format "UDX", which adds lexical type information and indicates which daughter node is the head, and a dictionary representation, which is useful for JSON serialization. All three are supported by PyDelphin.

Derivation trees have 3 types of nodes:

- root nodes, with only an entity name and a single child
- normal nodes, with 5 fields (below) and a list of children
  - *id* – an integer id given by the producer of the derivation
  - *entity* – rule or type name
  - *score* – a (MaxEnt) score for the current node's subtree
  - *start* – the character index of the left-most side of the tree
  - *end* – the character index of the right-most side of the tree
- terminal/left/lexical nodes, which contain the input tokens processed by that subtree

This module uses the `UdfNode` class for capturing root and normal nodes. Root nodes are expressed as a `UdfNode` whose `id` is `None`. For root nodes, all fields except `entity` and the list of daughters are expected to be `None`. Leaf nodes are simply an iterable of token information.

The `Derivation` class—itself a `UdfNode`—, has some tree-level operations defined, in particular the `Derivation.from_string()` method, which is used to read the serialized derivation into a Python object.

7.1 Loading Derivation Data

For loading a full derivation structure from either the UDF/UDX string representations or the dictionary representation, the `Derivation` class provides class methods to help with the decoding.

```
>>> from delphin import derivation
>>> d1 = derivation.Derivation.from_string(
...     '(1 entity-name 1 0 1 ("token"))')
...
>>> d2 = derivation.Derivation.from_dict(
...     {'id': 1, 'entity': 'entity-name', 'score': 1,
...      'start': 0, 'end': 1, 'form': 'token'}]})
...
>>> d1 == d2
True
```

class delphin.derivation.**Derivation**(*id*, *entity*, *score=None*, *start=None*, *end=None*, *daughters=None*, *head=None*, *type=None*, *parent=None*)

Bases: `delphin.derivation.UdfNode`

A [incr tsdb()] derivation.

This class exists to facilitate the reading of UDF string serializations and dictionary representations (e.g., decoded from JSON). The resulting structure is otherwise equivalent to a `UdfNode`, and inherits all its methods.

classmethod from_dict(*d*)

Instantiate a `Derivation` from a dictionary representation.

The dictionary representation may come from the HTTP interface (see the ErgApi wiki) or from the `UdfNode.to_dict()` method. Note that in the former case, the JSON response should have already been decoded into a Python dictionary.

Parameters **d** (*dict*) – dictionary representation of a derivation

classmethod from_string(*s*)

Instantiate a `Derivation` from a UDF or UDX string representation.

The UDF/UDX representations are as output by a processor like the LKB or ACE, or from the `UdfNode.to_udf()` or `UdfNode.to_udx()` methods.

Parameters **s** (*str*) – UDF or UDX serialization

7.2 UDF/UDX Node Types

There are three different node Types

class delphin.derivation.**UdfNode**

Normal (non-leaf) nodes in the Unified Derivation Format.

Root nodes are just UdfNodes whose `id`, by convention, is `None`. The `daughters` list can composed of either UdfNodes or other objects (generally it should be uniformly one or the other). In the latter case, the `UdfNode` is a preterminal, and the daughters are terminal nodes.

Parameters

- **id** (*int*) – unique node identifier
- **entity** (*str*) – grammar entity represented by the node
- **score** (*float, optional*) – probability or weight of the node
- **start** (*int, optional*) – start position of tokens encompassed by the node
- **end** (*int, optional*) – end position of tokens encompassed by the node
- **daughters** (*list, optional*) – iterable of daughter nodes
- **head** (*bool, optional*) – `True` if the node is a syntactic head node
- **type** (*str, optional*) – grammar type name
- **parent** (`UdfNode`, *optional*) – parent node in derivation

id

the unique node identifier

entity

the grammar entity represented by the node

score

the probability or weight of to the node; for many processors, this will be the unnormalized MaxEnt score assigned to the whole subtree rooted by this node

start
the start position (in inter-word, or chart, indices) of the substring encompassed by this node and its daughters

end
the end position (in inter-word, or chart, indices) of the substring encompassed by this node and its daughters

type
the lexical type (available on preterminal UDX nodes)

is_root()
Return `True` if the node is a root node.

Note: This is not simply the top node; by convention, a node is a root if its `id` is `None`.

---

to_udf(*indent=1*)
Encode the node and its descendants in the UDF format.

Parameters **indent** (*int*) – the number of spaces to indent at each level

Returns *str* – the UDF-serialized string

to_udx(*indent=1*)
Encode the node and its descendants in the UDF export format.

Parameters **indent** (*int*) – the number of spaces to indent at each level

Returns *str* – the UDX-serialized string

to_dict(*fields=('start', 'head', 'form', 'type', 'entity', 'id', 'end', 'tokens', 'daughters', 'score')*, *labels=None*)
Encode the node as a dictionary suitable for JSON serialization.

Parameters

- **fields** – if given, this is a whitelist of fields to include on nodes (`daughters` and `form` are always shown)
- **labels** – optional label annotations to embed in the derivation dict; the value is a list of lists matching the structure of the derivation (e.g., `["S" ["NP" ["NNS" ["Dogs"]]] ["VP" ["VBZ" ["bark"]]]]`)

Returns *dict* – the dictionary representation of the structure

basic_entity()
Return the entity without the lexical type information.

In the export (UDX) format, lexical types follow entities of preterminal nodes, joined by an at-sign (`@`). In regular UDF or non-preterminal nodes, this will just return the entity string.

Deprecated since version 0.5.1: Use *entity*

is_head()
Return `True` if the node is a head.

A node is a head if it is marked as a head in the UDX format or it has no siblings. `False` is returned if the node is known to not be a head (has a sibling that is a head). Otherwise it is indeterminate whether the node is a head, and `None` is returned.

is_root()
Return `True` if the node is a root node.

Note: This is not simply the top node; by convention, a node is a root if its `id` is `None`.

lexical_type()
Return the lexical type of a preterminal node.

In export (UDX) format, lexical types follow entities of preterminal nodes, joined by an at-sign (`@`). In regular UDF or non-preterminal nodes, this will return None.

Deprecated since version 0.5.1: Use *type*

preterminals()
Return the list of preterminals (i.e. lexical grammar-entities).

terminals()
Return the list of terminals (i.e. lexical units).

class delphin.derivation.**UdfTerminal**
Terminal nodes in the Unified Derivation Format.

The *form* field is always set, but *tokens* may be `None`.

See: http://moin.delph-in.net/ItsdbDerivations

Parameters

- **form** (*str*) – surface form of the terminal
- **tokens** (*list, optional*) – iterable of tokens
- **parent** (*UdfNode, optional*) – parent node in derivation

form
the surface form of the terminal

tokens
the list of tokens

is_root()
Return `False` (as a `UdfTerminal` is never a root).

This function is provided for convenience, so one does not need to check if `isinstance(n, UdfNode)` before testing if the node is a root.

to_udf(*indent=1*)
Encode the node and its descendants in the UDF format.

Parameters **indent** (*int*) – the number of spaces to indent at each level

Returns *str* – the UDF-serialized string

to_udx(*indent=1*)
Encode the node and its descendants in the UDF export format.

Parameters **indent** (*int*) – the number of spaces to indent at each level

Returns *str* – the UDX-serialized string

to_dict(*fields=('start', 'head', 'form', 'type', 'entity', 'id', 'end', 'tokens', 'daughters', 'score'), labels=None*)
Encode the node as a dictionary suitable for JSON serialization.

Parameters

- **fields** – if given, this is a whitelist of fields to include on nodes (`daughters` and `form` are always shown)

- **labels** – optional label annotations to embed in the derivation dict; the value is a list of lists matching the structure of the derivation (e.g., `["S" ["NP" ["NNS" ["Dogs"]]] ["VP" ["VBZ" ["bark"]]]]`)

Returns *dict* – the dictionary representation of the structure

is_root()

Return `False` (as a `UdfTerminal` is never a root).

This function is provided for convenience, so one does not need to check if `isinstance(n, UdfNode)` before testing if the node is a root.

class delphin.derivation.**UdfToken**

A token represenatation in derivations.

Token data are not formally nodes, but do have an `id`. Most *`UdfTerminal`* nodes will only have one UdfToken, but multi-word entities (e.g. "ad hoc") will have more than one.

Parameters

- **id** (*int*) – token identifier
- **tfs** (*str*) – the feature structure for the token

id

the token identifier

form

the feature structure for the token

CHAPTER 8

delphin.exceptions

exception delphin.exceptions.**PyDelphinException**(**args*, ***kwargs*)
The base class for PyDelphin exceptions.

exception delphin.exceptions.**PyDelphinWarning**(**args*, ***kwargs*)
The base class for PyDelphin warnings.

exception delphin.exceptions.**ItsdbError**(**args*, ***kwargs*)
Raised when there is an error processing a [incr tsdb()] profile.

exception delphin.exceptions.**REPPError**(**args*, ***kwargs*)
Raised when there is an error in tokenizing with REPP.

exception delphin.exceptions.**TdlError**(**args*, ***kwargs*)
Raised when there is an error in processing TDL.

exception delphin.exceptions.**TdlParsingError**(**args*, ***kwargs*)
Raised when parsing TDL text fails.

exception delphin.exceptions.**TdlWarning**(**args*, ***kwargs*)
Raised when parsing unsupported TDL features.

exception delphin.exceptions.**TSQLSyntaxError**(**args*, ***kwargs*)

exception delphin.exceptions.**XmrsError**(**args*, ***kwargs*)
Raised when there is an error processing *MRS objects.

exception delphin.exceptions.**XmrsWarning**(**args*, ***kwargs*)
Warning class for *MRS processing.

CHAPTER 9

delphin.extra

Non-core but generally useful modules.

This package contains modules that are generally useful but don't fit in the regular package hierarchy, and are either tightly integrated with PyDelphin or are too small to warrant creating a separate repository.

9.1 MRS and TDL Syntax Highlighting

Pygments-based highlighting lexers for DELPH-IN formats.

class `delphin.extra.highlight.`**`SimpleMrsLexer`**(***options*)

A Pygments-based Lexer for the SimpleMRS serialization format.

`get_tokens_unprocessed`(*text*)

Split `text` into (tokentype, text) pairs.

`stack` is the inital stack (default: `['root']`)

class `delphin.extra.highlight.`**`TdlLexer`**(***options*)

A Pygments-based Lexer for Typed Description Language.

See also:

- Pygments: http://pygments.org/

9.2 LaTeX Serialization

Generate LaTeX snippets for rendering DELPH-IN data.

`delphin.extra.latex.`**`dmrs_tikz_dependency`**(*xs*, ***kwargs*)

Return a LaTeX document with each Xmrs in *xs* rendered as DMRSs.

DMRSs use the `tikz-dependency` package for visualization.

See also:

- `tikz-dependency` package for LaTeX: https://ctan.org/pkg/tikz-dependency

CHAPTER 10

delphin.interfaces

Interface modules for external data providers.

PyDelphin interfaces manage the communication between PyDelphin and external DELPH-IN data providers. A data provider could be a local process, such as the ACE parser/generator, or a remote service, such as the DELPH-IN RESTful web server. The interfaces send requests to the providers, then receive and interpret the response. The interfaces may also detect and deserialize supported DELPH-IN formats.

10.1 delphin.interfaces.ace

See also:

See *Using ACE from PyDelphin* for a more user-friendly introduction.

An interface for the ACE processor.

This module provides classes and functions for managing interactive communication with an open ACE process.

Note: ACE is required for the functionality in this module, but it is not included with PyDelphin. Pre-compiled binaries are available for Linux and MacOS: http://sweaglesw.org/linguistics/ace/

For installation instructions, see: http://moin.delph-in.net/AceInstall

The `AceParser`, `AceTransferer`, and `AceGenerator` classes are used for parsing, transferring, and generating with ACE. All are subclasses of `AceProcess`, which connects to ACE in the background, sends it data via its stdin, and receives responses via its stdout. Responses from ACE are interpreted so the data is more accessible in Python.

Warning: Instantiating `AceParser`, `AceTransferer`, or `AceGenerator` opens ACE in a subprocess, so take care to close the process (`AceProcess.close()`) when finished or, alternatively, instantiate the class in a context manager.

Interpreted responses are stored in a dictionary-like *ParseResponse* object. When queried as a dictionary, these objects return the raw response strings. When queried via its methods, the PyDelphin models of the data are returned. The response objects may contain a number of `ParseResult` objects. These objects similarly provide raw-string access via dictionary keys and PyDelphin-model access via methods. Here is an example of parsing a sentence with *AceParser*:

```
>>> with AceParser('erg-1214-x86-64-0.9.24.dat') as parser:
...     response = parser.interact('Cats sleep.')
...     print(response.result(0)['mrs'])
...     print(response.result(0).mrs())
...
[ LTOP: h0 INDEX: e2 [ e SF: prop TENSE: pres MOOD: indicative PROG: - PERF: - ]␣
↪RELS: < [ udef_q<0:4> LBL: h4 ARG0: x3 [ x PERS: 3 NUM: pl IND: + ] RSTR: h5 BODY:␣
↪h6 ]  [ _cat_n_1<0:4> LBL: h7 ARG0: x3 ]  [ _sleep_v_1<5:11> LBL: h1 ARG0: e2 ARG1:␣
↪x3 ] > HCONS: < h0 qeq h1 h5 qeq h7 > ]
<Xmrs object (udef cat sleep) at 139880862399696>
```

Functions exist for non-interactive communication with ACE: *parse()* and *parse_from_iterable()* open and close an *AceParser* instance; *transfer()* and *transfer_from_iterable()* open and close an *AceTransferer* instance; and *generate()* and *generate_from_iterable()* open and close an *AceGenerator* instance. Note that these functions open a new ACE subprocess every time they are called, so if you have many items to process, it is more efficient to use *parse_from_iterable()*, *transfer_from_iterable()*, or *generate_from_iterable()* than the single-item versions, or to interact with the *AceProcess* subclass instances directly.

10.1.1 Basic Usage

The following module funtions are the simplest way to interact with ACE, although for larger or more interactive jobs it is suggested to use an *AceProcess* subclass instance.

`delphin.interfaces.ace.`**`compile`**(*cfg_path*, *out_path*, *executable=None*, *env=None*, *log=None*)

Use ACE to compile a grammar.

Parameters

- **cfg_path** (*str*) – the path to the ACE config file
- **out_path** (*str*) – the path where the compiled grammar will be written
- **executable** (*str, optional*) – the path to the ACE binary; if `None`, the `ace` command will be used
- **env** (*dict, optional*) – environment variables to pass to the ACE subprocess
- **log** (*file, optional*) – if given, the file, opened for writing, or stream to write ACE's stdout and stderr compile messages

`delphin.interfaces.ace.`**`parse`**(*grm*, *datum*, ***kwargs*)

Parse sentence *datum* with ACE using grammar *grm*.

Parameters

- **grm** (*str*) – path to a compiled grammar image
- **datum** (*str*) – the sentence to parse
- ****kwargs** – additional keyword arguments to pass to the AceParser

Returns `ParseResponse`

Example

```
>>> response = ace.parse('erg.dat', 'Dogs bark.')
NOTE: parsed 1 / 1 sentences, avg 797k, time 0.00707s
```

`delphin.interfaces.ace.`**`parse_from_iterable`**(*grm*, *data*, ***kwargs*)

Parse each sentence in *data* with ACE using grammar *grm*.

Parameters

- **grm** (*str*) – path to a compiled grammar image
- **data** (*iterable*) – the sentences to parse
- ****kwargs** – additional keyword arguments to pass to the AceParser

Yields `ParseResponse`

Example

```
>>> sentences = ['Dogs bark.', 'It rained']
>>> responses = list(ace.parse_from_iterable('erg.dat', sentences))
NOTE: parsed 2 / 2 sentences, avg 723k, time 0.01026s
```

`delphin.interfaces.ace.`**`transfer`**(*grm*, *datum*, ***kwargs*)

Transfer from the MRS *datum* with ACE using grammar *grm*.

Parameters

- **grm** (*str*) – path to a compiled grammar image
- **datum** – source MRS as a SimpleMRS string
- ****kwargs** – additional keyword arguments to pass to the AceTransferer

Returns `ParseResponse`

`delphin.interfaces.ace.`**`transfer_from_iterable`**(*grm*, *data*, ***kwargs*)

Transfer from each MRS in *data* with ACE using grammar *grm*.

Parameters

- **grm** (*str*) – path to a compiled grammar image
- **data** (*iterable*) – source MRSs as SimpleMRS strings
- ****kwargs** – additional keyword arguments to pass to the AceTransferer

Yields `ParseResponse`

`delphin.interfaces.ace.`**`generate`**(*grm*, *datum*, ***kwargs*)

Generate from the MRS *datum* with ACE using *grm*.

Parameters

- **grm** (*str*) – path to a compiled grammar image
- **datum** – the SimpleMRS string to generate from
- ****kwargs** – additional keyword arguments to pass to the AceGenerator

Returns `ParseResponse`

`delphin.interfaces.ace.`**`generate_from_iterable`**(*grm*, *data*, ***kwargs*)

Generate from each MRS in *data* with ACE using grammar *grm*.

Parameters

- **grm** (*str*) – path to a compiled grammar image
- **data** (*iterable*) – MRSs as SimpleMRS strings
- ****kwargs** – additional keyword arguments to pass to the AceGenerator

Yields `ParseResponse`

10.1.2 Classes for Managing ACE Processes

The functions described in *Basic Usage* are useful for small jobs as they handle the input and then close the ACE process, but for more complicated or interactive jobs, directly interacting with an instance of an `AceProcess` sublass is recommended or required (e.g., in the case of [incr tsdb()] testsuite processing). The `AceProcess` class is where most methods are defined, but in practice the `AceParser`, `AceTransferer`, or `AceGenerator` subclasses are directly used.

class `delphin.interfaces.ace.`**AceProcess**(*grm*, *cmdargs=None*, *executable=None*, *env=None*, *tsdbinfo=True*, ***kwargs*)

Bases: `delphin.interfaces.base.Processor`

The base class for interfacing ACE.

This manages most subprocess communication with ACE, but does not interpret the response returned via ACE's stdout. Subclasses override the `receive()` method to interpret the task-specific response formats.

Parameters

- **grm** (*str*) – path to a compiled grammar image
- **cmdargs** (*list, optional*) – a list of command-line arguments for ACE; note that arguments and their values should be separate entries, e.g. `['-n', '5']`
- **executable** (*str, optional*) – the path to the ACE binary; if `None`, ACE is assumed to be callable via `ace`
- **env** (*dict*) – environment variables to pass to the ACE subprocess
- **tsdbinfo** (*bool*) – if `True` and ACE's version is compatible, all information ACE reports for [incr tsdb()] processing is gathered and returned in the response

ace_version

The version of the specified ACE binary.

close()

Close the ACE process and return the process's exit code.

interact(*datum*)

Send *datum* to ACE and return the response.

This is the recommended method for sending and receiving data to/from an ACE process as it reduces the chances of over-filling or reading past the end of the buffer. It also performs a simple validation of the input to help ensure that one complete item is processed at a time.

If input item identifiers need to be tracked throughout processing, see `process_item()`.

Parameters **datum** (*str*) – the input sentence or MRS

Returns `ParseResponse`

process_item(*datum*, *keys=None*)

Send *datum* to ACE and return the response with context.

The *keys* parameter can be used to track item identifiers through an ACE interaction. If the `task` member is set on the AceProcess instance (or one of its subclasses), it is kept in the response as well. :param datum: the input sentence or MRS :type datum: str :param keys: a mapping of item identifier names and values :type keys: dict

Returns `ParseResponse`

receive()

Return the stdout response from ACE.

> **Warning:** Reading beyond the last line of stdout from ACE can cause the process to hang while it waits for the next line. Use the `interact()` method for most data-processing tasks with ACE.

run_info

Contextual information about the the running process.

send(*datum*)

Send *datum* (e.g. a sentence or MRS) to ACE.

> **Warning:** Sending data without reading (e.g., via `receive()`) can fill the buffer and cause data to be lost. Use the `interact()` method for most data-processing tasks with ACE.

task = None

The name of the task performed by the processor (`'parse'`, `'transfer'`, or `'generate'`). This is useful when a function, such as `delphin.itsdb.TestSuite.process()`, accepts any `AceProcess` instance.

class delphin.interfaces.ace.**AceParser**(*grm*, *cmdargs=None*, *executable=None*, *env=None*, *tsdbinfo=True*, ***kwargs*)

Bases: `delphin.interfaces.ace.AceProcess`

A class for managing parse requests with ACE.

See `AceProcess` for initialization parameters.

class delphin.interfaces.ace.**AceTransferer**(*grm*, *cmdargs=None*, *executable=None*, *env=None*, *tsdbinfo=False*, ***kwargs*)

Bases: `delphin.interfaces.ace.AceProcess`

A class for managing transfer requests with ACE.

Note that currently the `tsdbinfo` parameter must be set to `False` as ACE is not yet able to provide detailed information for transfer results.

See `AceProcess` for initialization parameters.

class delphin.interfaces.ace.**AceGenerator**(*grm*, *cmdargs=None*, *executable=None*, *env=None*, *tsdbinfo=True*, ***kwargs*)

Bases: `delphin.interfaces.ace.AceProcess`

A class for managing realization requests with ACE.

See `AceProcess` for initialization parameters.

10.2 delphin.interfaces.rest

Client access to the HTTP ("RESTful") API for DELPH-IN data.

This module provides classes and functions for making requests to servers that implement the DELPH-IN web API described here:

http://moin.delph-in.net/ErgApi

Note: Requires `requests` (https://pypi.python.org/pypi/requests)

Basic access is available via the *parse()* and *parse_from_iterable()* functions:

```
>>> from delphin.interfaces import rest
>>> url = 'http://erg.delph-in.net/rest/0.9/'
>>> rest.parse('Abrams slept.', server=url)
ParseResponse({'input': 'Abrams slept.', 'tcpu': 0.05, ...
>>> rest.parse_from_iterable(['Abrams slept.', 'It rained.'], server=url)
<generator object parse_from_iterable at 0x7f546661c258>
```

If the `server` parameter is not provided to `parse()`, the default ERG server (as used above) is used by default. Request parameters (described at http://moin.delph-in.net/ErgApi) can be provided via the `params` argument.

These functions both instantiate and use the *DelphinRestClient* class, which manages the connections to a server. It can also be used directly:

```
>>> client = rest.DelphinRestClient(server=url)
>>> client.parse('Dogs chase cats.')
ParseResponse({'input': 'Dogs chase cats.', ...
```

The server responds with JSON data, which PyDelphin parses to a dictionary. The responses from *DelphinRestClient.parse()* are then wrapped in *ParseResponse* objects, which provide two methods for inspecting the results. The *ParseResponse.result()* method takes a parameter `i` and returns the *i*th result (0-indexed), and the *ParseResponse.results()* method returns the list of all results. The benefit of using these methods is that they wrap the result dictionary in a *ParseResult* object, which provides methods for automatically deserializing derivations, EDS, MRS, or DMRS data. For example:

```
>>> r = client.parse('Dogs chase cats', params={'mrs':'json'})
>>> r.result(0)
ParseResult({'result-id': 0, 'score': 0.5938, ...
>>> r.result(0)['mrs']
{'variables': {'h1': {'type': 'h'}, 'x6': ...
>>> r.result(0).mrs()
<Xmrs object (udef dog chase udef cat) at 140000394933248>
```

If PyDelphin does not support deserialization for a format provided by the server (e.g. LaTeX output), the original string would be returned by these methods (i.e. the same as via dict-access).

10.2.1 Basic Usage

`delphin.interfaces.rest.`**`parse`**(*input*, *server='http://erg.delph-in.net/rest/0.9/'*, *params=None*, *headers=None*)

Request a parse of *input* on *server* and return the response.

Parameters

- **input** (*str*) – sentence to be parsed
- **server** (*str*) – the url for the server (the default LOGON server is used by default)
- **params** (*dict*) – a dictionary of request parameters

- **headers** (*dict*) – a dictionary of additional request headers

Returns A ParseResponse containing the results, if the request was successful.

Raises `requests.HTTPError` – if the status code was not 200

`delphin.interfaces.rest.`**`parse_from_iterable`**(*inputs*, *server='http://erg.delphin.net/rest/0.9/'*, *params=None*, *headers=None*)

Request parses for all *inputs*.

Parameters

- **inputs** (*iterable*) – sentences to parse
- **server** (*str*) – the url for the server (the default LOGON server is used by default)
- **params** (*dict*) – a dictionary of request parameters
- **headers** (*dict*) – a dictionary of additional request headers

Yields ParseResponse objects for each successful response.

Raises `requests.HTTPError` – for the first response with a status code that is not 200

10.2.2 Client Class

class `delphin.interfaces.rest.`**`DelphinRestClient`**(*server='http://erg.delphin.net/rest/0.9/'*)

Bases: *delphin.interfaces.base.Processor*

A class for managing requests to a DELPH-IN web API server.

parse(*sentence*, *params=None*, *headers=None*)

Request a parse of *sentence* and return the response.

Parameters

- **sentence** (*str*) – sentence to be parsed
- **params** (*dict*) – a dictionary of request parameters
- **headers** (*dict*) – a dictionary of additional request headers

Returns A ParseResponse containing the results, if the request was successful.

Raises `requests.HTTPError` – if the status code was not 200

process_item(*datum*, *keys=None*, *params=None*, *headers=None*)

Send *datum* to the processor and return the result.

This method is a generic wrapper around a processor-specific processing method that keeps track of additional item and processor information. Specifically, if *keys* is provided, it is copied into the `keys` key of the response object, and if the processor object's `task` member is non-`None`, it is copied into the `task` key of the response. These help with keeping track of items when many are processed at once, and to help downstream functions identify what the process did.

Parameters

- **datum** – the item content to process
- **keys** – a mapping of item identifiers which will be copied into the response

10.3 Common Classes

class delphin.interfaces.base.**FieldMapper**

A class for mapping responses to [incr tsdb()] fields.

This class provides two methods for mapping responses to fields:

- **map() - takes a response and returns a list of (table, data)** tuples for the data in the response, as well as aggregating any necessary information
- **cleanup() - returns any (table, data) tuples resulting from** aggregated data over all runs, then clears this data

In addition, the `affected_tables` attribute should list the names of tables that become invalidated by using this FieldMapper to process a profile. Generally this is the list of tables that `map()` and `cleanup()` create records for, but it may also include those that rely on the previous set (e.g., treebanking preferences, etc.).

Alternative [incr tsdb()] schema can be handled by overriding these two methods and the __init__() method.

affected_tables

list of tables that are affected by the processing

cleanup()

Return aggregated (table, rowdata) tuples and clear the state.

map(*response*)

Process *response* and return a list of (table, rowdata) tuples.

class delphin.interfaces.base.**ParseResponse**

A wrapper around the response dictionary for more convenient access to results.

result(*i*)

Return a ParseResult object for the *i*th result.

results()

Return ParseResult objects for each result.

tokens(*tokenset='internal'*)

Deserialize and return a YyTokenLattice object for the initial or internal token set, if provided, from the YY format or the JSON-formatted data; otherwise return the original string.

Parameters **tokenset** (*str*) – return `'initial'` or `'internal'` tokens (default: `'internal'`)

Returns `YyTokenLattice`

class delphin.interfaces.base.**ParseResult**

A wrapper around a result dictionary to automate deserialization for supported formats. A ParseResult is still a dictionary, so the raw data can be obtained using dict access.

derivation()

Deserialize and return a Derivation object for UDF- or JSON-formatted derivation data; otherwise return the original string.

dmrs()

Deserialize and return a Dmrs object for JSON-formatted DMRS data; otherwise return the original string.

eds()

Deserialize and return an Eds object for native- or JSON-formatted EDS data; otherwise return the original string.

mrs()
Deserialize and return an Mrs object for simplemrs or JSON-formatted MRS data; otherwise return the original string.

tree()
Deserialize and return a labeled syntax tree. The tree data may be a standalone datum, or embedded in the derivation.

class delphin.interfaces.base.**Processor**
Base class for processors.

This class defines the basic interface for all PyDelphin processors, such as *AceProcess* and *DelphinRestClient*. It can also be used to define preprocessor wrappers of other processors such that it has the same interface, allowing it to be used, e.g., with *TestSuite.process()*.

task
name of the task the processor performs (e.g. "parse", "transfer", or "generate")

process_item(*datum*, *keys=None*)
Send *datum* to the processor and return the result.

This method is a generic wrapper around a processor-specific processing method that keeps track of additional item and processor information. Specifically, if *keys* is provided, it is copied into the `keys` key of the response object, and if the processor object's `task` member is non-`None`, it is copied into the `task` key of the response. These help with keeping track of items when many are processed at once, and to help downstream functions identify what the process did.

Parameters

- **datum** – the item content to process
- **keys** – a mapping of item identifiers which will be copied into the response

10.4 Wrapping a Processor for Preprocessing

The *Processor* class can be used to implement a preprocessor that maintains the same interface as the underlying processor. The following example wraps an *AceParser* instance of the English Resource Grammar with a *REPP* instance.

```
>>> from delphin.interfaces import ace, base
>>> from delphin import repp
>>>
>>> class REPPWrapper(base.Processor):
...     def __init__(self, cpu, rpp):
...         self.cpu = cpu
...         self.task = cpu.task
...         self.rpp = rpp
...     def process_item(self, datum, keys=None):
...         preprocessed_datum = str(self.rpp.tokenize(datum))
...         return self.cpu.process_item(preprocessed_datum, keys=keys)
...
>>> # The preprocessor can be used like a normal Processor:
>>> rpp = repp.REPP.from_config('../../grammars/erg/pet/repp.set')
>>> grm = '../../grammars/erg-1214-x86-64-0.9.27.dat'
>>> with ace.AceParser(grm, cmdargs=['-y']) as _cpu:
...     cpu = REPPWrapper(_cpu, rpp)
...     response = cpu.process_item('Abrams hired Browne.')
...     for result in response.results():
```

(continues on next page)

(continued from previous page)

```
...         print(result.mrs())
...
<Mrs object (proper named hire proper named) at 140488735960480>
<Mrs object (unknown compound udef named hire parg addressee proper named) at␣
↪140488736005424>
<Mrs object (unknown proper compound udef named hire parg named) at 140488736004864>
NOTE: parsed 1 / 1 sentences, avg 1173k, time 0.00986s
```

A similar technique could be used to manage external processes, such as MeCab for morphological segmentation of Japanese for Jacy. It could also be used to make a postprocessor, a backoff mechanism in case an input fails to parse, etc.

CHAPTER 11

delphin.itsdb

See also:

See *Working with [incr tsdb()] Testsuites* for a more user-friendly introduction

Classes and functions for working with [incr tsdb()] profiles.

The `itsdb` module provides classes and functions for working with [incr tsdb()] profiles (or, more generally, testsuites; see http://moin.delph-in.net/ItsdbTop). It handles the technical details of encoding and decoding records in tables, including escaping and unescaping reserved characters, pairing columns with their relational descriptions, casting types (such as `:integer`, etc.), and transparently handling gzipped tables, so that the user has a natural way of working with the data. Capabilities include:

- Reading and writing testsuites:

```
>>> from delphin import itsdb
>>> ts = itsdb.TestSuite('jacy/tsdb/gold/mrs')
>>> ts.write(path='mrs-copy')
```

- Selecting data by table name, record index, and column name or index:

```
>>> items = ts['item']            # get the items table
>>> rec = items[0]                # get the first record
>>> rec['i-input']                # input sentence of the first item
'    '
>>> rec[0]                        # values are cast on index retrieval
11
>>> rec.get('i-id')               # and on key retrieval
11
>>> rec.get('i-id', cast=False)   # unless cast=False
'11'
```

- Selecting data as a query (note that types are cast by default):

```
>>> next(ts.select('item:i-id@i-input@i-date'))  # query testsuite
[11, '    ', datetime.datetime(2006, 5, 28, 0, 0)]
```

(continues on next page)

(continued from previous page)

```
>>> next(items.select('i-id@i-input@i-date'))     # query table
[11, '    ', datetime.datetime(2006, 5, 28, 0, 0)]
```

- In-memory modification of testsuite data:

```
>>> # desegment each sentence
>>> for record in ts['item']:
...     record['i-input'] = ''.join(record['i-input'].split())
...
>>> ts['item'][0]['i-input']
''
```

- Joining tables

```
>>> joined = itsdb.join(ts['parse'], ts['result'])
>>> next(joined.select('i-id@mrs'))
[11, '[ LTOP: h1 INDEX: e2 [ e TENSE: PAST ...']
```

- Processing data with ACE (results are stored in memory)

```
>>> from delphin.interfaces import ace
>>> with ace.AceParser('jacy.dat') as cpu:
...     ts.process(cpu)
...
NOTE: parsed 126 / 135 sentences, avg 3167k, time 1.87536s
>>> ts.write('new-profile')
```

This module covers all aspects of [incr tsdb()] data, from `Relations` files and `Field` descriptions to `Record`, `Table`, and full `TestSuite` classes. `TestSuite` is the most user-facing interface, and it makes it easy to load the tables of a testsuite into memory, inspect its contents, modify or create data, and write the data to disk.

By default, the `itsdb` module expects testsuites to use the standard [incr tsdb()] schema. Testsuites are always read and written according to the associated or specified relations file, but other things, such as default field values and the list of "core" tables, are defined for the standard schema. It is, however, possible to define non-standard schemata for particular applications, and most functions will continue to work. One notable exception is the `TestSuite.process()` method, for which a new `FieldMapper` class must be defined.

11.1 Overview of [incr tsdb()] Testsuites

[incr tsdb()] testsuites are directories containing a `relations` file (see *Relations Files and Field Descriptions*) and a file for each table in the database. The typical testsuite contains these files:

```
testsuite/
  analysis  fold             item-set   parse       relations  run    tree
  decision  item             output     phenomenon  result     score  update
  edge      item-phenomenon  parameter  preference  rule       set
```

PyDelphin has three classes for working with [incr tsdb()] testsuite databases:

- `TestSuite` – The entire testsuite (or directory)
- `Table` – A table (or file) in a testsuite
- `Record` – A row (or line) in a table

class delphin.itsdb.**TestSuite**(*path=None*, *relations=None*, *encoding='utf-8'*)

A [incr tsdb()] testsuite database.

Parameters

- **path** – the path to the testsuite's directory
- **relations** (*Relations*, str) – the database schema; either a *Relations* object or a path to a relations file; if not given, the relations file under *path* will be used
- **encoding** – the character encoding of the files in the testsuite

encoding

character encoding used when reading and writing tables

Type str

relations

database schema

Type *Relations*

exists(*table=None*)

Return True if the testsuite or a table exists on disk.

If *table* is None, this method returns True if the TestSuite.path is specified and points to an existing directory containing a valid relations file. If *table* is given, the function returns True if, in addition to the above conditions, the table exists as a file (even if empty). Otherwise it returns False.

process(*cpu*, *selector=None*, *source=None*, *fieldmapper=None*, *gzip=None*, *buffer_size=1000*)

Process each item in a [incr tsdb()] testsuite

If the testsuite is attached to files on disk, the output records will be flushed to disk when the number of new records in a table is *buffer_size*. If the testsuite is not attached to files or *buffer_size* is set to None, records are kept in memory and not flushed to disk.

Parameters

- **cpu** (*Processor*) – processor interface (e.g., *AceParser*)
- **selector** (*str*) – data specifier to select a single table and column as processor input (e.g., "item:i-input")
- **source** (*TestSuite*, *Table*) – testsuite or table from which inputs are taken; if None, use self
- **fieldmapper** (*FieldMapper*) – object for mapping response fields to [incr tsdb()] fields; if None, use a default mapper for the standard schema
- **gzip** – compress non-empty tables with gzip
- **buffer_size** (*int*) – number of output records to hold in memory before flushing to disk; ignored if the testsuite is all in-memory; if None, do not flush to disk

Examples

```
>>> ts.process(ace_parser)
>>> ts.process(ace_generator, 'result:mrs', source=ts2)
```

reload()

Discard temporary changes and reload the database from disk.

select (*arg*, *cols=None*, *mode='list'*)

Select columns from each row in the table.

The first parameter, *arg*, may either be a table name or a data specifier. If the former, the *cols* parameter selects the columns from the table. If the latter, *cols* is left unspecified and both the table and columns are taken from the data specifier; e.g., `select('item:i-id@i-input')` is equivalent to `select('item', ('i-id', 'i-input'))`.

See select_rows() for a description of how to use the *mode* parameter.

Parameters

- **arg** – a table name, if *cols* is specified, otherwise a data specifier
- **cols** – an iterable of Field (column) names
- **mode** – how to return the data

size (*table=None*)

Return the size, in bytes, of the testsuite or *table*.

If *table* is `None`, return the size of the whole testsuite (i.e., the sum of the table sizes). Otherwise, return the size of *table*.

Notes

- If the file is gzipped, it returns the compressed size.
- Only tables on disk are included.

write (*tables=None*, *path=None*, *relations=None*, *append=False*, *gzip=None*)

Write the testsuite to disk.

Parameters

- **tables** – a name or iterable of names of tables to write, or a Mapping of table names to table data; if `None`, all tables will be written
- **path** – the destination directory; if `None` use the path assigned to the TestSuite
- **relations** – a *`Relations`* object or path to a relations file to be used when writing the tables
- **append** – if `True`, append to rather than overwrite tables
- **gzip** – compress non-empty tables with gzip

Examples

```
>>> ts.write(path='new/path')
>>> ts.write('item')
>>> ts.write(['item', 'parse', 'result'])
>>> ts.write({'item': item_rows})
```

class `delphin.itsdb.`**Table** (*fields*, *records=None*)

A [incr tsdb()] table.

Instances of this class contain a collection of rows with the data stored in the database. Generally a Table will be created by a *`TestSuite`* object for a database, but a Table can also be instantiated individually by the *`Table.from_file()`* class method, and the relations file in the same directory is used to get the schema. Tables can also be constructed entirely in-memory and separate from a testsuite via the standard `Table()` constructor.

Tables have two modes: **attached** and **detached**. Attached tables are backed by a file on disk (whether as part of a testsuite or not) and only store modified records in memory—all unmodified records are retrieved from disk. Therefore, iterating over a table is more efficient than random-access. Attached files use significantly less memory than detached tables but also require more processing time. Detached tables are entirely stored in memory and are not backed by a file. They are useful for the programmatic construction of testsuites (including for unit tests) and other operations where high-speed random-access is required. See the `attach()` and `detach()` methods for more information. The `is_attached()` method is useful for determining the mode of a table.

Parameters

- **fields** – the Relation schema for this table
- **records** – the collection of Record objects containing the table data

name

table name

Type str

fields

table schema

Type `Relation`

path

if attached, the path to the file containing the table data; if detached it is `None`

Type str

encoding

the character encoding of the attached table file; if detached it is `None`

Type str

classmethod from_file(*path*, *fields=None*, *encoding='utf-8'*)

Instantiate a Table from a database file.

This method instantiates a table attached to the file at *path*. The file will be opened and traversed to determine the number of records, but the contents will not be stored in memory unless they are modified.

Parameters

- **path** – the path to the table file
- **fields** – the Relation schema for the table (loaded from the relations file in the same directory if not given)
- **encoding** – the character encoding of the file at *path*

write(*records=None*, *path=None*, *fields=None*, *append=False*, *gzip=None*)

Write the table to disk.

The basic usage has no arguments and writes the table's data to the attached file. The parameters accommodate a variety of use cases, such as using *fields* to refresh a table to a new schema or *records* and *append* to incrementally build a table.

Parameters

- **records** – an iterable of `Record` objects to write; if `None` the table's existing data is used
- **path** – the destination file path; if `None` use the path of the file attached to the table
- **fields** (`Relation`) – table schema to use for writing, otherwise use the current one
- **append** – if `True`, append rather than overwrite

- **gzip** – compress with gzip if non-empty

Examples

```
>>> table.write()
>>> table.write(results, path='new/path/result')
```

commit()

Commit changes to disk if attached.

This method helps normalize the interface for detached and attached tables and makes writing attached tables a bit more efficient. For detached tables nothing is done, as there is no notion of changes, but neither is an error raised (unlike with `write()`). For attached tables, if all changes are new records, the changes are appended to the existing file, and otherwise the whole file is rewritten.

attach(*path*, *encoding='utf-8'*)

Attach the Table to the file at *path*.

Attaching a table to a file means that only changed records are stored in memory, which greatly reduces the memory footprint of large profiles at some cost of performance. Tables created from `Table.from_file()` or from an attached `TestSuite` are automatically attached. Attaching a file does not immediately flush the contents to disk; after attaching the table must be separately written to commit the in-memory data.

A non-empty table will fail to attach to a non-empty file to avoid data loss when merging the contents. In this case, you may delete or clear the file, clear the table, or attach to another file.

Parameters

- **path** – the path to the table file
- **encoding** – the character encoding of the files in the testsuite

detach()

Detach the table from a file.

Detaching a table reads all data from the file and places it in memory. This is useful when constructing or significantly manipulating table data, or when more speed is needed. Tables created by the default constructor are detached.

When detaching, only unmodified records are loaded from the file; any uncommited changes in the Table are left as-is.

> **Warning:** Very large tables may consume all available RAM when detached. Expect the in-memory table to take up about twice the space of an uncompressed table on disk, although this may vary by system.

is_attached()

Return `True` if the table is attached to a file.

list_changes()

Return a list of modified records.

This is only applicable for attached tables.

Returns A list of `(row_index, record)` tuples of modified records

Raises `delphin.exceptions.ItsdbError` – when called on a detached table

append(*record*)
Add *record* to the end of the table.

Parameters **record** – a `Record` or other iterable containing column values

extend(*records*)
Add each record in *records* to the end of the table.

Parameters **record** – an iterable of `Record` or other iterables containing column values

select(*cols*, *mode='list'*)
Select columns from each row in the table.

See `select_rows()` for a description of how to use the *mode* parameter.

Parameters

- **cols** – an iterable of Field (column) names
- **mode** – how to return the data

class delphin.itsdb.**Record**(*fields*, *iterable*)
A row in a [incr tsdb()] table.

Parameters

- **fields** – the Relation schema for the table of this record
- **iterable** – an iterable containing the data for the record

fields
table schema

Type `Relation`

classmethod from_dict(*fields*, *mapping*)
Create a Record from a dictionary of field mappings.

The *fields* object is used to determine the column indices of fields in the mapping.

Parameters

- **fields** – the Relation schema for the table of this record
- **mapping** – a dictionary or other mapping from field names to column values

Returns a `Record` object

get(*key*, *default=None*, *cast=True*)
Return the field data given by field name *key*.

Parameters

- **key** – the field name of the data to return
- **default** – the value to return if *key* is not in the row

11.2 Relations Files and Field Descriptions

A "relations file" is a required file in [incr tsdb()] testsuites that describes the schema of the database. The file contains descriptions of each table and each field within the table. The first 9 lines of `run` table description is as follows:

```
run:
  run-id :integer :key                    # unique test run identifier
  run-comment :string                     # descriptive narrative
  platform :string                        # implementation platform (version)
  protocol :integer                       # [incr tsdb()] protocol version
  tsdb :string                            # tsdb(1) (version) used
  application :string                     # application (version) used
  environment :string                     # application-specific information
  grammar :string                         # grammar (version) used
  ...
```

In PyDelphin, there are three classes for modeling this information:

- `Relations` – the entire relations file schema
- `Relation` – the schema for a single table
- `Field` – a single field description

class `delphin.itsdb.`**Relations**(*tables*)

A [incr tsdb()] database schema.

Note: Use `from_file()` or `from_string()` for instantiating a Relations object.

Parameters **tables** – a list of (table, `Relation`) tuples

find(*fieldname*)

Return the list of tables that define the field *fieldname*.

classmethod from_file(*source*)

Instantiate Relations from a relations file.

classmethod from_string(*s*)

Instantiate Relations from a relations string.

items()

Return a list of (table, `Relation`) for each table.

path(*source*, *target*)

Find the path of id fields connecting two tables.

This is just a basic breadth-first-search. The relations file should be small enough to not be a problem.

Returns

list –

(table, fieldname) pairs describing the path from the source to target tables

Raises `delphin.exceptions.ItsdbError` – when no path is found

Example

```
>>> relations.path('item', 'result')
[('parse', 'i-id'), ('result', 'parse-id')]
>>> relations.path('parse', 'item')
[('item', 'i-id')]
```

(continues on next page)

(continued from previous page)

```
>>> relations.path('item', 'item')
[]
```

class delphin.itsdb.**Relation**

A [incr tsdb()] table schema.

Parameters

- **name** – the table name
- **fields** – a list of Field objects

index(*fieldname*)

Return the Field index given by *fieldname*.

keys()

Return the tuple of field names of key fields.

class delphin.itsdb.**Field**

A tuple describing a column in an [incr tsdb()] profile.

Parameters

- **name** (*str*) – the column name
- **datatype** (*str*) – ":string", ":integer", ":date", or ":float"
- **key** (*bool*) – True if the column is a key in the database
- **partial** (*bool*) – True if the column is a partial key
- **comment** (*str*) – a description of the column

default_value()

Get the default value of the field.

11.3 Utility Functions

delphin.itsdb.**join**(*table1*, *table2*, *on=None*, *how='inner'*, *name=None*)

Join two tables and return the resulting Table object.

Fields in the resulting table have their names prefixed with their corresponding table name. For example, when joining item and parse tables, the i-input field of the item table will be named item:i-input in the resulting Table. Pivot fields (those in *on*) are only stored once without the prefix.

Both inner and left joins are possible by setting the *how* parameter to inner and left, respectively.

> **Warning:** Both *table2* and the resulting joined table will exist in memory for this operation, so it is not recommended for very large tables on low-memory systems.

Parameters

- **table1** (*Table*) – the left table to join
- **table2** (*Table*) – the right table to join
- **on** (*str*) – the shared key to use for joining; if None, find shared keys using the schemata of the tables

- **how** (*str*) – the method used for joining ("inner" or "left")
- **name** (*str*) – the name assigned to the resulting table

delphin.itsdb.**match_rows**(*rows1*, *rows2*, *key*, *sort_keys=True*)

Yield triples of `(value, left_rows, right_rows)` where `left_rows` and `right_rows` are lists of rows that share the same column value for *key*. This means that both *rows1* and *rows2* must have a column with the same name *key*.

> **Warning:** Both *rows1* and *rows2* will exist in memory for this operation, so it is not recommended for very large tables on low-memory systems.

Parameters

- **rows1** – a *Table* or list of *Record* objects
- **rows2** – a *Table* or list of *Record* objects
- **key** (*str*) – the column name on which to match
- **sort_keys** (*bool*) – if `True`, yield matching rows sorted by the matched key instead of the original order

delphin.itsdb.**select_rows**(*cols*, *rows*, *mode='list'*, *cast=True*)

Yield data selected from rows.

It is sometimes useful to select a subset of data from a profile. This function selects the data in *cols* from *rows* and yields it in a form specified by *mode*. Possible values of *mode* are:

mode	description	example `['i-id', 'i-wf']`
`'list'` (default)	a list of values	`[10, 1]`
`'dict'`	col to value map	`{'i-id': 10,'i-wf': 1}`
`'row'`	[incr tsdb()] row	`'10@1'`

Parameters

- **cols** – an iterable of column names to select data for
- **rows** – the rows to select column data from
- **mode** – the form yielded data should take
- **cast** – if `True`, cast column values to their datatype (requires *rows* to be *Record* objects)

Yields Selected data in the form specified by *mode*.

delphin.itsdb.**make_row**(*row*, *fields*)

Encode a mapping of column name to values into a [incr tsdb()] profile line. The *fields* parameter determines what columns are used, and default values are provided if a column is missing from the mapping.

Parameters

- **row** – a mapping of column names to values
- **fields** – an iterable of *Field* objects

Returns A [incr tsdb()]-encoded string

`delphin.itsdb.escape(string)`

Replace any special characters with their [incr tsdb()] escape sequences. The characters and their escape sequences are:

```
@         -> \s
(newline) -> \n
\         -> \\
```

Also see `unescape()`

Parameters **string** – the string to escape

Returns The escaped string

`delphin.itsdb.unescape(string)`

Replace [incr tsdb()] escape sequences with the regular equivalents. Also see `escape()`.

Parameters **string** (`str`) – the escaped string

Returns The string with escape sequences replaced

`delphin.itsdb.decode_row(line, fields=None)`

Decode a raw line from a profile into a list of column values.

Decoding involves splitting the line by the field delimiter (`"@"` by default) and unescaping special characters. If *fields* is given, cast the values into the datatype given by their respective Field object.

Parameters

- **line** – a raw line from a [incr tsdb()] profile.
- **fields** – a list or Relation object of Fields for the row

Returns A list of column values.

`delphin.itsdb.encode_row(fields)`

Encode a list of column values into a [incr tsdb()] profile line.

Encoding involves escaping special characters for each value, then joining the values into a single string with the field delimiter (`"@"` by default). It does not fill in default values (see make_row()).

Parameters **fields** – a list of column values

Returns A [incr tsdb()]-encoded string

`delphin.itsdb.get_data_specifier(string)`

Return a tuple (table, col) for some [incr tsdb()] data specifier. For example:

```
item              -> ('item', None)
item:i-input      -> ('item', ['i-input'])
item:i-input@i-wf -> ('item', ['i-input', 'i-wf'])
:i-input          -> (None, ['i-input'])
(otherwise)       -> (None, None)
```

11.4 Deprecated

The following are remnants of the old functionality that will be removed in a future version, but remain for now to aid in the transition.

class `delphin.itsdb.ItsdbProfile(path, relations=None, filters=None, applicators=None, index=True, cast=False, encoding='utf-8')`

A [incr tsdb()] profile, analyzed and ready for reading or writing.

Parameters

- **path** – The path of the directory containing the profile
- **filters** – A list of tuples [(table, cols, condition)] such that only rows in table where condition(row, row[col]) evaluates to a non-false value are returned; filters are tested in order for a table.
- **applicators** – A list of tuples [(table, cols, function)] which will be used when reading rows from a table—the function will be applied to the contents of the column cell in the table. For each table, each column-function pair will be applied in order. Applicators apply after the filters.
- **index** – If `True`, indices are created based on the keys of each table.
- **cast** – if `True`, automatically cast data into the type defined by its relation field (e.g., :integer)

Deprecated since version v0.7.0.

add_applicator(*table*, *cols*, *function*)

Add an applicator. When reading *table*, rows in *table* will be modified by apply_rows().

Parameters

- **table** – The table to apply the function to.
- **cols** – The columns in *table* to apply the function on.
- **function** – The applicator function.

add_filter(*table*, *cols*, *condition*)

Add a filter. When reading *table*, rows in *table* will be filtered by filter_rows().

Parameters

- **table** – The table the filter applies to.
- **cols** – The columns in *table* to filter on.
- **condition** – The filter function.

exists(*table=None*)

Return True if the profile or a table exist.

If *table* is `None`, this function returns True if the root directory exists and contains a valid relations file. If *table* is given, the function returns True if the table exists as a file (even if empty). Otherwise it returns False.

join(*table1*, *table2*, *key_filter=True*)

Yield rows from a table built by joining *table1* and *table2*. The column names in the rows have the original table name prepended and separated by a colon. For example, joining tables 'item' and 'parse' will result in column names like 'item:i-input' and 'parse:parse-id'.

read_raw_table(*table*)

Yield rows in the [incr tsdb()] *table*. A row is a dictionary mapping column names to values. Data from a profile is decoded by decode_row(). No filters or applicators are used.

read_table(*table*, *key_filter=True*)

Yield rows in the [incr tsdb()] *table* that pass any defined filters, and with values changed by any applicators. If no filters or applicators are defined, the result is the same as from ItsdbProfile.read_raw_table().

select(*table*, *cols*, *mode='list'*, *key_filter=True*)

Yield selected rows from *table*. This method just calls select_rows() on the rows read from *table*.

size(*table=None*)

Return the size, in bytes, of the profile or *table*.

If *table* is `None`, this function returns the size of the whole profile (i.e. the sum of the table sizes). Otherwise, it returns the size of *table*.

Note: if the file is gzipped, it returns the compressed size.

write_profile(*profile_directory*, *relations_filename=None*, *key_filter=True*, *append=False*, *gzip=None*)

Write all tables (as specified by the relations) to a profile.

Parameters

- **profile_directory** – The directory of the output profile
- **relations_filename** – If given, read and use the relations at this path instead of the current profile's relations
- **key_filter** – If True, filter the rows by keys in the index
- **append** – If `True`, append profile data to existing tables in the output profile directory
- **gzip** – If `True`, compress tables using `gzip`. Table filenames will have `.gz` appended. If `False`, only write out text files. If `None`, use whatever the original file was.

write_table(*table*, *rows*, *append=False*, *gzip=False*)

Encode and write out *table* to the profile directory.

Parameters

- **table** – The name of the table to write
- **rows** – The rows to write to the table
- **append** – If `True`, append the encoded rows to any existing data.
- **gzip** – If `True`, compress the resulting table with `gzip`. The table's filename will have `.gz` appended.

class delphin.itsdb.**ItsdbSkeleton**(*path*, *relations=None*, *filters=None*, *applicators=None*, *index=True*, *cast=False*, *encoding='utf-8'*)

A [incr tsdb()] skeleton, analyzed and ready for reading or writing.

See `ItsdbProfile` for initialization parameters.

Deprecated since version v0.7.0.

delphin.itsdb.**get_relations**(*path*)

Parse the relations file and return a Relations object that describes the database structure.

Note: for backward-compatibility only; use Relations.from_file()

Parameters **path** – The path of the relations file.

Returns A dictionary mapping a table name to a list of Field tuples.

Deprecated since version v0.7.0.

delphin.itsdb.**default_value**(*fieldname*, *datatype*)

Return the default value for a column.

If the column name (e.g. *i-wf*) is defined to have an idiosyncratic value, that value is returned. Otherwise the default value for the column's datatype is returned.

Parameters

- **fieldname** – the column name (e.g. `i-wf`)

- **datatype** – the datatype of the column (e.g. `:integer`)

Returns The default value for the column.

Deprecated since version v0.7.0.

`delphin.itsdb.`**`make_skeleton`**(*path*, *relations*, *item_rows*, *gzip=False*)

Instantiate a new profile skeleton (only the relations file and item file) from an existing relations file and a list of rows for the item table. For standard relations files, it is suggested to have, as a minimum, the `i-id` and `i-input` fields in the item rows.

Parameters

- **path** – the destination directory of the skeleton—must not already exist, as it will be created
- **relations** – the path to the relations file
- **item_rows** – the rows to use for the item file
- **gzip** – if `True`, the item file will be compressed

Returns An ItsdbProfile containing the skeleton data (but the profile data will already have been written to disk).

Raises *`delphin.exceptions.ItsdbError`* – if the destination directory could not be created.

Deprecated since version v0.7.0.

`delphin.itsdb.`**`filter_rows`**(*filters*, *rows*)

Yield rows matching all applicable filters.

Filter functions have binary arity (e.g. `filter(row, col)`) where the first parameter is the dictionary of row data, and the second parameter is the data at one particular column.

Parameters

- **filters** – a tuple of (cols, filter_func) where filter_func will be tested (filter_func(row, col)) for each col in cols where col exists in the row
- **rows** – an iterable of rows to filter

Yields Rows matching all applicable filters

Deprecated since version v0.7.0.

`delphin.itsdb.`**`apply_rows`**(*applicators*, *rows*)

Yield rows after applying the applicator functions to them.

Applicators are simple unary functions that return a value, and that value is stored in the yielded row. E.g. `row[col] = applicator(row[col])`. These are useful to, e.g., cast strings to numeric datatypes, to convert formats stored in a cell, extract features for machine learning, and so on.

Parameters

- **applicators** – a tuple of (cols, applicator) where the applicator will be applied to each col in cols
- **rows** – an iterable of rows for applicators to be called on

Yields Rows with specified column values replaced with the results of the applicators

Deprecated since version v0.7.0.

CHAPTER 12

delphin.mrs

This module contains classes and methods related to Minimal Recursion Semantics [MRS]. In addition to MRS, there are the related formalisms Robust Minimal Recursion Semantics [RMRS], Elementary Dependency Structures [EDS], and Dependency Minimal Recursion Semantics [DMRS]. As a convenience, *MRS refers to the collection of MRS and related formalisms (so "MRS" then refers to the original formalism), and PyDelphin accordingly defines `Xmrs` as the common subclass for the various formalisms.

Users will interact mostly with `Xmrs` objects, but will not often instantiate them directly. Instead, they are created by serializing one of the various formats (such as `delphin.mrs.simplemrs`, `delphin.mrs.mrx`, or :mod;'delphin.mrs.dmrx'). No matter what serialization format (or formalism) is used to load a *MRS structure, it will be stored the same way in memory, so any queries or actions taken on these structures will use the same methods.

12.1 delphin.mrs.compare

`delphin.mrs.compare.`**`compare_bags`**(*testbag*, *goldbag*, *count_only=True*)

Compare two bags of Xmrs objects, returning a triple of (unique in test, shared, unique in gold).

Parameters

- **`testbag`** – An iterable of Xmrs objects to test.
- **`goldbag`** – An iterable of Xmrs objects to compare against.
- **`count_only`** – If True, the returned triple will only have the counts of each; if False, a list of Xmrs objects will be returned for each (using the ones from testbag for the shared set)

Returns A triple of (unique in test, shared, unique in gold), where each of the three items is an integer count if the count_only parameter is True, or a list of Xmrs objects otherwise.

`delphin.mrs.compare.`**`isomorphic`**(*q*, *g*, *check_varprops=True*)

Return `True` if Xmrs objects *q* and *g* are isomorphic.

Isomorphicity compares the predicates of an Xmrs, the variable properties of their predications (if `check_varprops=True`), constant arguments, and the argument structure between predications. Node IDs and Lnk values are ignored.

Parameters

- **q** – the left Xmrs to compare
- **g** – the right Xmrs to compare
- **check_varprops** – if `True`, make sure variable properties are equal for mapped predications

12.2 delphin.mrs.components

Classes and functions for working with the components of *MRS objects.

12.2.1 Variables and Predicates

`delphin.mrs.components.`**`sort_vid_split`**`(vs)`

Split a valid variable string into its variable sort and id.

Examples

```
>>> sort_vid_split('h3')
('h', '3')
>>> sort_vid_split('ref-ind12')
('ref-ind', '12')
```

`delphin.mrs.components.`**`var_sort`**`(v)`

Return the sort of a valid variable string.

Examples

```
>>> var_sort('h3')
'h'
>>> var_sort('ref-ind12')
'ref-ind'
```

`delphin.mrs.components.`**`var_id`**`(v)`

Return the integer id of a valid variable string.

Examples

```
>>> var_id('h3')
3
>>> var_id('ref-ind12')
12
```

class `delphin.mrs.components.`**`Pred`**

A semantic predicate.

Abstract predicates don't begin with an underscore, and they generally are defined as types in a grammar. **Surface** predicates always begin with an underscore (ignoring possible quotes), and are often defined as strings in a lexicon.

In PyDelphin, Preds are equivalent if they have the same lemma, pos, and sense, and are both abstract or both surface preds. Other factors are ignored for comparison, such as their being surface-, abstract-, or real-preds, whether they are quoted or not, whether they end with `_rel` or not, or differences in capitalization. Hashed Pred objects (e.g., in a dict or set) also use the normalized form. However, unlike with equality comparisons, Pred-formatted strings are not treated as equivalent in a hash.

Parameters

- **type** – the type of predicate; valid values are Pred.ABSTRACT, Pred.REALPRED, and Pred.SURFACE, although in practice Preds are instantiated via classmethods that select the type
- **lemma** – the lemma of the predicate
- **pos** – the part-of-speech; a single, lowercase character
- **sense** – the (often omitted) sense of the predicate

Returns a Pred object

type
predicate type (Pred.ABSTRACT, Pred.REALPRED, and Pred.SURFACE)

lemma
lemma component of the predicate

pos
part-of-speech component of the predicate

sense
sense component of the predicate

Example

Preds are compared using their string representations. Surrounding quotes (double or single) are ignored, and capitalization doesn't matter. In addition, preds may be compared directly to their string representations:

```
>>> p1 = Pred.surface('_dog_n_1_rel')
>>> p2 = Pred.realpred(lemma='dog', pos='n', sense='1')
>>> p3 = Pred.abstract('dog_n_1_rel')
>>> p1 == p2
True
>>> p1 == '_dog_n_1_rel'
True
>>> p1 == p3
False
```

classmethod abstract(*predstr*)
Instantiate a Pred from its symbol string.

classmethod grammarpred(*predstr*)
Instantiate a Pred from its symbol string.

is_quantifier()
Return `True` if the predicate has a quantifier part-of-speech.

Deprecated since v0.6.0

classmethod realpred(*lemma*, *pos*, *sense=None*)
Instantiate a Pred from its components.

short_form()

Return the pred string without quotes or a `_rel` suffix.

The short form is the same as the normalized form from *normalize_pred_string()*.

Example

```
>>> p = Pred.surface('"_cat_n_1_rel"')
>>> p.short_form()
'_cat_n_1'
```

classmethod string_or_grammar_pred(*predstr*)

Instantiate a Pred from either its surface or abstract symbol.

classmethod stringpred(*predstr*)

Instantiate a Pred from its quoted string representation.

classmethod surface(*predstr*)

Instantiate a Pred from its quoted string representation.

classmethod surface_or_abstract(*predstr*)

Instantiate a Pred from either its surface or abstract symbol.

delphin.mrs.components.**is_valid_pred_string**(*predstr*)

Return `True` if *predstr* is a valid predicate string.

Examples

```
>>> is_valid_pred_string('"_dog_n_1_rel"')
True
>>> is_valid_pred_string('_dog_n_1')
True
>>> is_valid_pred_string('_dog_noun_1')
False
>>> is_valid_pred_string('dog_noun_1')
True
```

delphin.mrs.components.**normalize_pred_string**(*predstr*)

Normalize the predicate string *predstr* to a conventional form.

This makes predicate strings more consistent by removing quotes and the `_rel` suffix, and by lowercasing them.

Examples

```
>>> normalize_pred_string('"_dog_n_1_rel"')
'_dog_n_1'
>>> normalize_pred_string('_dog_n_1')
'_dog_n_1'
```

delphin.mrs.components.**split_pred_string**(*predstr*)

Split *predstr* and return the (lemma, pos, sense, suffix) components.

Examples

```
>>> Pred.split_pred_string('_dog_n_1_rel')
('dog', 'n', '1', 'rel')
>>> Pred.split_pred_string('quant_rel')
('quant', None, None, 'rel')
```

12.2.2 MRS and RMRS Components

class `delphin.mrs.components.`**`ElementaryPredication`**

An MRS elementary predication (EP).

EPs combine a predicate with various structural semantic properties. They must have a `nodeid`, `pred`, and `label`. Arguments and other properties are optional. Note nodeids are not a formal property of MRS (unlike DMRS, or the "anchors" of RMRS), but they are required for Pydelphin to uniquely identify EPs in an *`Xmrs`*. Intrinsic arguments (`ARG0`) are not required, but they are important for many semantic operations, and therefore it is a good idea to include them.

Parameters

- **`nodeid`** – a nodeid
- **`pred`** (*`Pred`*) – semantic predicate
- **`label`** (*`str`*) – scope handle
- **`args`** (*`dict, optional`*) – mapping of roles to values
- **`lnk`** (`Lnk`, optional) – surface alignment
- **`surface`** (*`str, optional`*) – surface string
- **`base`** (*`str, optional`*) – base form

`nodeid`

a nodeid

`pred`

semantic predicate

Type *`Pred`*

`label`

scope handle

Type str

`args`

mapping of roles to values

Type dict

`lnk`

surface alignment

Type `Lnk`

`surface`

surface string

Type str

base
base form

Type str

cfrom
surface alignment starting position

Type int

cto
surface alignment ending position

Type int

carg
The value of the constant argument.

intrinsic_variable
The value of the intrinsic argument (likely `ARG0`).

is_quantifier()
Return `True` if this is a quantifier predication.

iv
A synonym for *`ElementaryPredication.intrinsic_variable`*

delphin.mrs.components.**elementarypredication**(*xmrs*, *nodeid*)
Retrieve the elementary predication with the given nodeid.

Parameters **nodeid** – nodeid of the EP to return

Returns *`ElementaryPredication`*

Raises `KeyError` – if no EP matches

delphin.mrs.components.**elementarypredications**(*xmrs*)
Return the list of *`ElementaryPredication`* objects in *xmrs*.

class delphin.mrs.components.**HandleConstraint**
A relation between two handles.

Parameters

- **hi** (*str*) – hi handle (hole) of the constraint
- **relation** (*str*) – relation of the constraint (nearly always `"qeq"`, but `"lheq"` and `"outscopes"` are also valid)
- **lo** (*str*) – lo handle (label) of the constraint

hi
hi handle (hole) of the constraint

Type str

relation
relation of the constraint

Type str

lo
lo handle (label) of the constraint

Type str

`delphin.mrs.components.hcons(xmrs)`
Return the list of all HandleConstraints in *xmrs*.

class `delphin.mrs.components.IndividualConstraint`
A relation between two variables.

Parameters

- **left** (*str*) – left variable of the constraint
- **relation** (*str*) – relation of the constraint
- **right** (*str*) – right variable of the constraint

left
left variable of the constraint

Type str

relation
relation of the constraint

Type str

right
right variable of the constraint

Type str

`delphin.mrs.components.icons(xmrs)`
Return the list of all IndividualConstraints in *xmrs*.

12.2.3 DMRS and EDS Components

class `delphin.mrs.components.Node`
A DMRS node.

Nodes are very simple predications for DMRSs. Nodes don't have arguments or labels like *ElementaryPredication* objects, but they do have a property for CARGs and contain their variable sort and properties in `sortinfo`.

Parameters

- **nodeid** – node identifier
- **pred** (*Pred*) – semantic predicate
- **sortinfo** (*dict, optional*) – mapping of morphosyntactic properties and values; the `cvarsort` property is specified in this mapping
- **lnk** (`Lnk`, optional) – surface alignment
- **surface** (*str, optional*) – surface string
- **base** (*str, optional*) – base form
- **carg** (*str, optional*) – constant argument string

Attributes

pred
semantic predicate

Type *Pred*

sortinfo
mapping of morphosyntactic properties and values; the `cvarsort` property is specified in this mapping

Type dict

lnk
surface alignment

Type `Lnk`

surface
surface string

Type str

base
base form

Type str

carg
constant argument string

Type str

cfrom
surface alignment starting position

Type int

cto
surface alignment ending position

Type int

cvarsort
The "variable" type of the predicate.

Note: DMRS does not use variables, but it is useful to indicate whether a node is an individual, eventuality, etc., so this property encodes that information.

is_quantifier()
Return `True` if the Node's predicate appears to be a quantifier.

Deprecated since v0.6.0

properties
Morphosemantic property mapping.

Unlike *sortinfo*, this does not include `cvarsort`.

`delphin.mrs.components.`**nodes**(*xmrs*)
Return the list of Nodes for *xmrs*.

class `delphin.mrs.components.`**Link**
DMRS-style dependency link.

Links are a way of representing arguments without variables. A Link encodes a start and end node, the role name, and the scopal relationship between the start and end (e.g. label equality, qeq, etc).

Parameters

- **start** – nodeid of the start of the Link

- **end** – nodeid of the end of the Link
- **rargname** (*str*) – role of the argument
- **post** (*str*) – "post-slash label" indicating the scopal relationship between the start and end of the Link; possible values are `NEQ`, `EQ`, `HEQ`, and `H`

start
nodeid of the start of the Link

end
nodeid of the end of the Link

rargname
role of the argument

Type str

post
"post-slash label" indicating the scopal relationship between the start and end of the Link

Type str

`delphin.mrs.components.`**links**(*xmrs*)
Return the list of Links for the *xmrs*.

12.3 delphin.mrs.dmrx

DMRX (XML for DMRS) serialization and deserialization.

`delphin.mrs.dmrx.`**dump**(*destination*, *ms*, *single=False*, *properties=True*, *pretty_print=False*, ***kwargs*)
Serialize Xmrs objects to DMRX and write to a file

Parameters

- **destination** – filename or file object where data will be written
- **ms** – an iterator of Xmrs objects to serialize (unless the *single* option is `True`)
- **single** – if `True`, treat *ms* as a single Xmrs object instead of as an iterator
- **properties** – if `False`, suppress variable properties
- **pretty_print** – if `True`, add newlines and indentation

`delphin.mrs.dmrx.`**dumps**(*ms*, *single=False*, *properties=True*, *pretty_print=False*, ***kwargs*)
Serialize an Xmrs object to a DMRX representation

Parameters

- **ms** – an iterator of Xmrs objects to serialize (unless the *single* option is `True`)
- **single** – if `True`, treat *ms* as a single Xmrs object instead of as an iterator
- **properties** – if `False`, suppress variable properties
- **pretty_print** – if `True`, add newlines and indentation

Returns a DMRX string representation of a corpus of Xmrs

`delphin.mrs.dmrx.`**load**(*fh*, *single=False*)
Deserialize DMRX from a file (handle or filename)

Parameters

- **fh** (*str, file*) – input filename or file object
- **single** – if `True`, only return the first read Xmrs object

Returns a generator of Xmrs objects (unless the *single* option is `True`)

`delphin.mrs.dmrx.loads`(*s*, *single=False*)

Deserialize DMRX string representations

Parameters

- **s** (*str*) – a DMRX string
- **single** (*bool*) – if `True`, only return the first Xmrs object

Returns a generator of Xmrs objects (unless *single* is `True`)

12.4 delphin.mrs.eds

Elementary Dependency Structures (EDS).

class `delphin.mrs.eds.Eds`(*top=None*, *nodes=None*, *edges=None*)

An Elementary Dependency Structure (EDS) instance.

EDS are semantic structures deriving from MRS, but they are not interconvertible with MRS and therefore do not share a common superclass (viz, *Xmrs*) in PyDelphin, although this may change in a future version. EDS shares some common features with DMRS, so this class borrows some DMRS elements, such as *Nodes*.

Parameters

- **top** – nodeid of the top node in the graph
- **nodes** – iterable of *Nodes*
- **edges** – iterable of (start, role, end) triples

edges(*nodeid*)

Return the edges starting at *nodeid*.

classmethod from_dict(*d*)

Decode a dictionary, as from *to_dict()*, into an Eds object.

classmethod from_triples(*triples*)

Decode triples, as from *to_triples()*, into an Eds object.

classmethod from_xmrs(*xmrs*, *predicate_modifiers=False*, ***kwargs*)

Instantiate an Eds from an Xmrs (lossy conversion).

Parameters

- **xmrs** (*Xmrs*) – Xmrs instance to convert from
- **predicate_modifiers** (*function, bool*) – function that is called as `func(xmrs, deps)` after finding the basic dependencies (`deps`), returning a mapping of predicate-modifier dependencies; the form of `deps` and the returned mapping are `{head: [(role, dependent)]}`; if *predicate_modifiers* is `True`, the function is created using *non_argument_modifiers()* as: `non_argument_modifiers(role="ARG1", connecting=True); if *predicate_modifiers* is `False`, only the basic dependencies are returned

node(*nodeid*)

Return the node whose `nodeid` is *nodeid*.

nodeids()
Return the list of nodeids.

nodes()
Return the list of nodes.

to_dict(*properties=True*)
Encode the Eds as a dictionary suitable for JSON serialization.

to_triples(*short_pred=True*, *properties=True*)
Encode the Eds as triples suitable for PENMAN serialization.

delphin.mrs.eds.**dump**(*destination*, *ms*, *single=False*, *properties=False*, *pretty_print=True*, *show_status=False*, *predicate_modifiers=False*, ***kwargs*)
Serialize Xmrs objects to Eds and write to a file

Parameters

- **destination** – filename or file object where data will be written
- **ms** – an iterator of *Xmrs* objects to serialize (unless the *single* option is `True`)
- **single** (*bool*) – if `True`, treat *ms* as a single *Xmrs* object instead of as an iterator
- **properties** (*bool*) – if `False`, suppress variable properties
- **pretty_print** (*bool*) – if `True`, add newlines and indentation
- **show_status** (*bool*) – if `True`, annotate disconnected graphs and nodes

delphin.mrs.eds.**dumps**(*ms*, *single=False*, *properties=False*, *pretty_print=True*, *show_status=False*, *predicate_modifiers=False*, ***kwargs*)
Serialize an Xmrs object to a Eds representation

Parameters

- **ms** – an iterator of *Xmrs* objects to serialize (unless the *single* option is `True`)
- **single** (*bool*) – if `True`, treat *ms* as a single *Xmrs* object instead of as an iterator
- **properties** (*bool*) – if `False`, suppress variable properties
- **pretty_print** (*bool*) – if `True`, add newlines and indentation
- **show_status** (*bool*) – if `True`, annotate disconnected graphs and nodes

Returns an *Eds* string representation of a corpus of Xmrs

delphin.mrs.eds.**load**(*fh*, *single=False*)
Deserialize *Eds* from a file (handle or filename)

Parameters

- **fh** (*str, file*) – input filename or file object
- **single** (*bool*) – if `True`, only return the first Xmrs object

Returns a generator of *Eds* objects (unless the *single* option is `True`)

delphin.mrs.eds.**loads**(*s*, *single=False*)
Deserialize *Eds* string representations

Parameters

- **s** (*str*) – Eds string
- **single** (*bool*) – if `True`, only return the first Xmrs object

Returns a generator of *Eds* objects (unless the *single* option is `True`)

`delphin.mrs.eds.`**`non_argument_modifiers`**(*role='ARG1', only_connecting=True*)

Return a function that finds non-argument modifier dependencies.

Parameters

- **role** (*str*) – the role that is assigned to the dependency
- **only_connecting** (*bool*) – if `True`, only return dependencies that connect separate components in the basic dependencies; if `False`, all non-argument modifier dependencies are included

Returns a function with signature `func(xmrs, deps)` that returns a mapping of non-argument modifier dependencies

Examples

The default function behaves like the LKB:

```
>>> func = non_argument_modifiers()
```

A variation is similar to DMRS's MOD/EQ links:

```
>>> func = non_argument_modifiers(role="MOD", only_connecting=False)
```

12.5 delphin.mrs.mrx

MRX (XML for MRS) serialization and deserialization.

`delphin.mrs.mrx.`**`dump`**(*destination, ms, single=False, properties=True, encoding='unicode', pretty_print=False, **kwargs*)

Serialize Xmrs objects to MRX and write to a file

Parameters

- **destination** – filename or file object where data will be written
- **ms** – an iterator of Xmrs objects to serialize (unless the *single* option is `True`)
- **single** – if `True`, treat *ms* as a single Xmrs object instead of as an iterator
- **properties** – if `False`, suppress variable properties
- **encoding** – the character encoding of the string prior writing to a file (generally "`unicode`" is desired)
- **pretty_print** – if `True`, add newlines and indentation

`delphin.mrs.mrx.`**`dumps`**(*ms, single=False, properties=True, encoding='unicode', pretty_print=False, **kwargs*)

Serialize an Xmrs object to a MRX representation

Parameters

- **ms** – an iterator of Xmrs objects to serialize (unless the *single* option is `True`)
- **single** – if `True`, treat *ms* as a single Xmrs object instead of as an iterator
- **properties** – if `False`, suppress variable properties
- **encoding** – the character encoding of the string ("`unicode`" returns a regular (unicode) string in Python 3 and a unicode string in Python 2)

- **pretty_print** – if `True`, add newlines and indentation

Returns a MRX string representation of a corpus of Xmrs

`delphin.mrs.mrx.load`(*fh*, *single=False*)
Deserialize MRX from a file (handle or filename)

Parameters

- **fh** (*str, file*) – input filename or file object
- **single** – if `True`, only return the first read Xmrs object

Returns a generator of Xmrs objects (unless the *single* option is `True`)

`delphin.mrs.mrx.loads`(*s*, *single=False*)
Deserialize MRX string representations

Parameters

- **s** (*str*) – a MRX string
- **single** (*bool*) – if `True`, only return the first Xmrs object

Returns a generator of Xmrs objects (unless *single* is `True`)

12.6 delphin.mrs.penman

Serialization functions for the PENMAN graph format.

Unlike other *MRS serializers, this one takes a *model* argument for the load(), loads(), dump(), and dumps() methods, which determines what the graph will look like. This is because DMRS and EDS (and possibly others in the future) yield different graph structures, but both can be encoded as PENMAN graphs. In this sense, it is somewhat like how JSON formatting of *MRS is handled in PyDelphin.

class `delphin.mrs.penman.XMRSCodec`(*indent=True*, *relation_sort=<function original_order>*)
A customized PENMAN codec class for *MRS data.

`delphin.mrs.penman.dump`(*destination*, *xs*, *model=None*, *properties=False*, *indent=True*, ***kwargs*)
Serialize Xmrs (or subclass) objects to PENMAN and write to a file.

Parameters

- **destination** – filename or file object
- **xs** – iterator of *Xmrs* objects to serialize
- **model** – Xmrs subclass used to get triples
- **properties** – if `True`, encode variable properties
- **indent** – if `True`, adaptively indent; if `False` or `None`, don't indent; if a non-negative integer N, indent N spaces per level

`delphin.mrs.penman.dumps`(*xs*, *model=None*, *properties=False*, *indent=True*, ***kwargs*)
Serialize Xmrs (or subclass) objects to PENMAN notation

Parameters

- **xs** – iterator of *Xmrs* objects to serialize
- **model** – Xmrs subclass used to get triples
- **properties** – if `True`, encode variable properties

- **indent** – if `True`, adaptively indent; if `False` or `None`, don't indent; if a non-negative integer N, indent N spaces per level

Returns the PENMAN serialization of *xs*

`delphin.mrs.penman.load`(*fh*, *model*)
Deserialize PENMAN graphs from a file (handle or filename)

Parameters

- **fh** – filename or file object
- **model** – Xmrs subclass instantiated from decoded triples

Returns a list of objects (of class *model*)

`delphin.mrs.penman.loads`(*s*, *model*)
Deserialize PENMAN graphs from a string

Parameters

- **s** (*str*) – serialized PENMAN graphs
- **model** – Xmrs subclass instantiated from decoded triples

Returns a list of objects (of class *model*)

12.7 delphin.mrs.prolog

Serialization functions for the Prolog format.

Example

```
>>> from delphin.interfaces import rest
>>> from delphin.mrs import prolog
>>> response = rest.parse('The dog sleeps soundly.', params={'mrs':'json'})
>>> print(prolog.dumps([response.result(0).mrs()], pretty_print=True))
psoa(h1,e3,
[rel('_the_q',h4,
    [attrval('ARG0',x6),
        attrval('RSTR',h7),
        attrval('BODY',h5)]),
rel('_dog_n_1',h8,
    [attrval('ARG0',x6)]),
rel('_sleep_v_1',h2,
    [attrval('ARG0',e3),
        attrval('ARG1',x6)]),
rel('_sound_a_1',h2,
    [attrval('ARG0',e9),
        attrval('ARG1',e3)])],
hcons([qeq(h1,h2),qeq(h7,h8)]))
```

`delphin.mrs.prolog.dump`(*destination*, *ms*, *single=False*, *pretty_print=False*, ***kwargs*)
Serialize Xmrs objects to the Prolog representation and write to a file.

Parameters

- **destination** – filename or file object where data will be written
- **ms** – an iterator of Xmrs objects to serialize (unless the *single* option is `True`)

- **single** – if `True`, treat *ms* as a single Xmrs object instead of as an iterator
- **pretty_print** – if `True`, add newlines and indentation

`delphin.mrs.prolog.dumps`(*ms*, *single=False*, *pretty_print=False*, ***kwargs*)

Serialize an Xmrs object to the Prolog representation

Parameters

- **ms** – an iterator of Xmrs objects to serialize (unless the *single* option is `True`)
- **single** – if `True`, treat *ms* as a single Xmrs object instead of as an iterator
- **pretty_print** – if `True`, add newlines and indentation

Returns the Prolog string representation of a corpus of Xmrs

12.8 delphin.mrs.query

Functions for inspecting and interpreting the structure of an Xmrs.

`delphin.mrs.query.select_nodeids`(*xmrs*, *iv=None*, *label=None*, *pred=None*)

Return the list of matching nodeids in *xmrs*.

Nodeids in *xmrs* match if their corresponding `ElementaryPredication` object matches its `intrinsic_variable` to *iv*, `label` to *label*, and `pred` to *pred*. The *iv*, *label*, and *pred* filters are ignored if they are `None`.

Parameters

- **xmrs** (`Xmrs`) – semantic structure to query
- **iv** (`str, optional`) – intrinsic variable to match
- **label** (`str, optional`) – label to match
- **pred** (str, `Pred`, optional) – predicate to match

Returns *list* – matching nodeids

`delphin.mrs.query.select_nodes`(*xmrs*, *nodeid=None*, *pred=None*)

Return the list of matching nodes in *xmrs*.

DMRS `nodes` for *xmrs* match if their `nodeid` matches *nodeid* and `pred` matches *pred*. The *nodeid* and *pred* filters are ignored if they are `None`.

Parameters

- **xmrs** (`Xmrs`) – semantic structure to query
- **nodeid** (`optional`) – DMRS nodeid to match
- **pred** (str, `Pred`, optional) – predicate to match

Returns *list* – matching nodes

`delphin.mrs.query.select_eps`(*xmrs*, *nodeid=None*, *iv=None*, *label=None*, *pred=None*)

Return the list of matching elementary predications in *xmrs*.

`ElementaryPredication` objects for *xmrs* match if their `nodeid` matches *nodeid*, `intrinsic_variable` matches *iv*, `label` matches *label*, and `pred` to *pred*. The *nodeid*, *iv*, *label*, and *pred* filters are ignored if they are `None`.

Parameters

- **xmrs** (*Xmrs*) – semantic structure to query
- **nodeid** (*optional*) – nodeid to match
- **iv** (*str, optional*) – intrinsic variable to match
- **label** (*str, optional*) – label to match
- **pred** (str, *Pred*, optional) – predicate to match

Returns *list* – matching elementary predications

delphin.mrs.query.**select_args**(*xmrs*, *nodeid=None*, *rargname=None*, *value=None*)

Return the list of matching (nodeid, role, value) triples in *xmrs*.

Predication arguments in *xmrs* match if the `nodeid` of the *ElementaryPredication* they are arguments of match *nodeid*, their role matches *rargname*, and their value matches *value*. The *nodeid*, *rargname*, and *value* filters are ignored if they are `None`.

Note: The *value* filter matches the variable, handle, or constant that is the overt value for the argument. If you want to find arguments that target a particular nodeid, look into *Xmrs.incoming_args()*. If you want to match a target value to its resolved object, see *find_argument_target()*.

Parameters

- **xmrs** (*Xmrs*) – semantic structure to query
- **nodeid** (*optional*) – nodeid to match
- **rargname** (*str, optional*) – role name to match
- **value** (*str, optional*) – argument value to match

Returns *list* – matching arguments as (nodeid, role, value) triples

delphin.mrs.query.**select_links**(*xmrs*, *start=None*, *end=None*, *rargname=None*, *post=None*)

Return the list of matching links for *xmrs*.

Link objects for *xmrs* match if their `start` matches *start*, `end` matches *end*, `rargname` matches *rargname*, and `post` matches *post*. The *start*, *end*, *rargname*, and *post* filters are ignored if they are `None`.

Parameters

- **xmrs** (*Xmrs*) – semantic structure to query
- **start** (*optional*) – link start nodeid to match
- **end** (*optional*) – link end nodeid to match
- **rargname** (*str, optional*) – role name to match
- **post** (*str, optional*) – Link post-slash label to match

Returns *list* – matching links

delphin.mrs.query.**select_hcons**(*xmrs*, *hi=None*, *relation=None*, *lo=None*)

Return the list of matching HCONS for *xmrs*.

HandleConstraint objects for *xmrs* match if their `hi` matches *hi*, `relation` matches *relation*, and `lo` matches *lo*. The *hi*, *relation*, and *lo* filters are ignored if they are `None`.

Parameters

- **xmrs** (*Xmrs*) – semantic structure to query

- **hi** (*str, optional*) – hi handle (hole) to match
- **relation** (*str, optional*) – handle constraint relation to match
- **lo** (*str, optional*) – lo handle (label) to match

Returns *list* – matching HCONS

`delphin.mrs.query.`**`select_icons`**(*xmrs*, *left=None*, *relation=None*, *right=None*)

Return the list of matching ICONS for *xmrs*.

`IndividualConstraint` objects for *xmrs* match if their `left` matches *left*, `relation` matches *relation*, and `right` matches *right*. The *left*, *relation*, and *right* filters are ignored if they are `None`.

Parameters

- **xmrs** (*Xmrs*) – semantic structure to query
- **left** (*str, optional*) – left variable to match
- **relation** (*str, optional*) – individual constraint relation to match
- **right** (*str, optional*) – right variable to match

Returns *list* – matching ICONS

`delphin.mrs.query.`**`find_argument_target`**(*xmrs*, *nodeid*, *rargname*)

Return the target of an argument (rather than just the variable).

Note: If the argument value is an intrinsic variable whose target is an EP that has a quantifier, the non-quantifier EP's nodeid will be returned. With this nodeid, one can then use `Xmrs.nodeid()` to get its quantifier's nodeid.

Parameters

- **xmrs** (*Xmrs*) – semantic structure to use
- **nodeid** – nodeid of the argument.
- **rargname** – role name of the argument.

Returns

The object that is the target of the argument. Possible values include:

Argument value	e.g.	Target
intrinsic variable	x4	nodeid; of the EP with the IV
hole variable	h0	nodeid; HCONS's labelset head
label	h1	nodeid; label's labelset head
unbound variable	i3	the variable itself
constant	"IBM"	the constant itself

`delphin.mrs.query.`**`find_subgraphs_by_preds`**(*xmrs*, *preds*, *connected=None*)

Yield subgraphs matching a list of predicates.

Predicates may match multiple EPs/nodes in the *xmrs*, meaning that more than one subgraph is possible. Also, predicates in *preds* match in number, so if a predicate appears twice in *preds*, there will be two matching EPs/nodes in each subgraph.

Parameters

- **xmrs** (*Xmrs*) – semantic structure to use

- **preds** – iterable of predicates to include in subgraphs
- **connected** (*bool, optional*) – if `True`, all yielded subgraphs must be connected, as determined by *Xmrs.is_connected()*.

Yields A *Xmrs* object for each subgraphs found.

`delphin.mrs.query.`**`intrinsic_variables`**(*xmrs*)
Return the list of all intrinsic variables in *xmrs*

`delphin.mrs.query.`**`bound_variables`**(*xmrs*)
Return the list of all bound variables in *xmrs*

`delphin.mrs.query.`**`in_labelset`**(*xmrs*, *nodeids*, *label=None*)
Test if all nodeids share a label.

Parameters

- **nodeids** – iterable of nodeids
- **label** (*str, optional*) – the label that all nodeids must share

Returns *bool* – `True` if all nodeids share a label, otherwise `False`

12.9 delphin.mrs.semi

Semantic Interface (SEM-I)

Semantic interfaces (SEM-Is) describe the inventory of semantic components in a grammar, including variables, properties, roles, and predicates. This information can be used for validating semantic structures or for filling out missing information in incomplete representations.

See also:

- Wiki on SEM-I: http://moin.delph-in.net/SemiRfc

`delphin.mrs.semi.`**`load`**(*fn*)
Read the SEM-I beginning at the filename *fn* and return the SemI.

Parameters **fn** – the filename of the top file for the SEM-I. Note: this must be a filename and not a file-like object.

Returns The SemI defined by *fn*

class `delphin.mrs.semi.`**`SemI`**(*variables=None*, *properties=None*, *roles=None*, *predicates=None*)
A semantic interface.

SEM-Is describe the semantic inventory for a grammar. These include the variable types, valid properties for variables, valid roles for predications, and a lexicon of predicates with associated roles.

Parameters

- **variables** – a mapping of (var, Variable)
- **properties** – a mapping of (prop, Property)
- **roles** – a mapping of (role, Role)
- **predicates** – a mapping of (pred, Predicate)

classmethod from_dict(*d*)
Instantiate a SemI from a dictionary representation.

to_dict()
Return a dictionary representation of the SemI.

class delphin.mrs.semi.**Predicate**
An MRS predicate description.

classmethod from_dict(*d*)
Instantiate a Predicate from a dictionary representation.

to_dict()
Return a dictionary representation of the Predicate.

class delphin.mrs.semi.**Property**
An MRS morphosemantic property description.

classmethod from_dict(*d*)
Instantiate a Property from a dictionary representation.

to_dict()
Return a dictionary representation of the Property.

class delphin.mrs.semi.**Role**
An MRS role description.

classmethod from_dict(*d*)
Instantiate a Role from a dictionary representation.

properties
Properties constrained by the Role.

to_dict()
Return a dictionary representation of the Role.

class delphin.mrs.semi.**Variable**
An MRS variable description.

classmethod from_dict(*d*)
Instantiate a Variable from a dictionary representation.

properties
Properties constrained by the Variable.

to_dict()
Return a dictionary representation of the Variable.

12.10 delphin.mrs.simpledmrs

Serialization for the SimpleDMRS format.

Note that this format is provided by PyDelphin and not defined anywhere, so it should only be used for user's convenience and not used as an interchange format or for other practical purposes. It was created with human legibility in mind (e.g. for investigating DMRSs at the command line, because XML (DMRX) is not easy to read). Deserialization is not provided.

12.11 delphin.mrs.simplemrs

Serialization functions for the SimpleMRS format.

`delphin.mrs.simplemrs.dump`(*destination*, *ms*, *single=False*, *version=1.1*, *properties=True*, *pretty_print=False*, *color=False*, ***kwargs*)

Serialize Xmrs objects to SimpleMRS and write to a file

Parameters

- **destination** – filename or file object where data will be written
- **ms** – an iterator of Xmrs objects to serialize (unless the *single* option is `True`)
- **single** – if `True`, treat *ms* as a single Xmrs object instead of as an iterator
- **properties** – if `False`, suppress variable properties
- **pretty_print** – if `True`, add newlines and indentation
- **color** – if `True`, colorize the output with ANSI color codes

`delphin.mrs.simplemrs.dumps`(*ms*, *single=False*, *version=1.1*, *properties=True*, *pretty_print=False*, *color=False*, ***kwargs*)

Serialize an Xmrs object to a SimpleMRS representation

Parameters

- **ms** – an iterator of Xmrs objects to serialize (unless the *single* option is `True`)
- **single** – if `True`, treat *ms* as a single Xmrs object instead of as an iterator
- **properties** – if `False`, suppress variable properties
- **pretty_print** – if `True`, add newlines and indentation
- **color** – if `True`, colorize the output with ANSI color codes

Returns a SimpleMrs string representation of a corpus of Xmrs

`delphin.mrs.simplemrs.load`(*fh*, *single=False*, *version=1.1*, *strict=False*, *errors='warn'*)

Deserialize SimpleMRSs from a file (handle or filename)

Parameters

- **fh** (*str, file*) – input filename or file object
- **single** – if `True`, only return the first read Xmrs object
- **strict** – deprecated; a `True` value is the same as `errors='strict'`, and a `False` value is the same as `errors='warn'`
- **errors** – if `'strict'`, ill-formed MRSs raise an error; if `'warn'`, raise a warning instead; if `'ignore'`, do not warn or raise errors for ill-formed MRSs

Returns a generator of Xmrs objects (unless the *single* option is `True`)

`delphin.mrs.simplemrs.loads`(*s*, *single=False*, *version=1.1*, *strict=False*, *errors='warn'*)

Deserialize SimpleMRS string representations

Parameters

- **s** (*str*) – a SimpleMRS string
- **single** (*bool*) – if `True`, only return the first Xmrs object

Returns a generator of Xmrs objects (unless *single* is `True`)

`delphin.mrs.simplemrs.serialize`(*ms*, *version=1.1*, *properties=True*, *pretty_print=False*, *color=False*)

Serialize an MRS structure into a SimpleMRS string.

`delphin.mrs.simplemrs.tokenize(string)`
Split the SimpleMrs string into tokens.

12.12 delphin.mrs.vpm

Variable property mapping (VPM).

Variable property mappings (VPMs) convert grammar-internal variables (e.g. `event5`) to the grammar-external form (e.g. `e5`), and also map variable properties (e.g. `PNG: 1pl` might map to `PERS: 1` and `NUM: pl`).

See also:

- Wiki about VPM: http://moin.delph-in.net/RmrsVpm

`delphin.mrs.vpm.load(source, semi=None)`
Read a variable-property mapping from *source* and return the VPM.

Parameters

- **source** – a filename or file-like object containing the VPM definitions
- **semi** (*SemI*, optional) – if provided, it is passed to the VPM constructor

Returns a *VPM* instance

class `delphin.mrs.vpm.VPM(typemap, propmap, semi=None)`
A variable-property mapping.

This class contains the rules for mapping variable properties from the grammar-internal definitions to grammar-external ones, and back again.

Parameters

- **typemap** – an iterable of (src, OP, tgt) iterables
- **propmap** – an iterable of (featset, valmap) tuples, where featmap is a tuple of two lists: (source_features, target_features); and valmap is a list of value tuples: (source_values, OP, target_values)
- **semi** (*SemI*, optional) – if provided, this is used for more sophisticated value comparisons

apply(*var*, *props*, *reverse=False*)
Apply the VPM to variable *var* and properties *props*.

Parameters

- **var** – a variable
- **props** – a dictionary mapping properties to values
- **reverse** – if `True`, apply the rules in reverse (e.g. from grammar-external to grammar-internal forms)

Returns a tuple (v, p) of the mapped variable and properties

12.13 delphin.mrs.xmrs

Classes and functions for general *MRS processing.

class delphin.mrs.xmrs.**Dmrs**(*nodes=None*, *links=None*, *top=None*, *index=None*, *xarg=None*, *lnk=None*, *surface=None*, *identifier=None*)

Construct an *Xmrs* using DMRS components.

Dependency Minimal Recursion Semantics (DMRS) have a list of Node objects and a list of Link objects. There are no variables or handles, so these will need to be created in order to make an Xmrs object. The *top* node may be set directly via a parameter or may be implicitly set via a Link from the special nodeid 0. If both are given, the link is ignored. The *index* and *xarg* nodes may only be set via parameters.

Parameters

- **nodes** – an iterable of Node objects
- **links** – an iterable of Link objects
- **top** – the scopal top node
- **index** – the non-scopal top node
- **xarg** – the external argument node
- **lnk** – the Lnk object associating the MRS to the surface form
- **surface** – the surface string
- **identifier** – a discourse-utterance id

Example:

```
>>> rain = Node(10000, Pred.surface('_rain_v_1_rel'),
>>>             sortinfo={'cvarsort': 'e'})
>>> ltop_link = Link(0, 10000, post='H')
>>> d = Dmrs([rain], [ltop_link])
```

classmethod from_dict(*d*)

Decode a dictionary, as from *to_dict()*, into a Dmrs object.

classmethod from_triples(*triples*, *remap_nodeids=True*)

Decode triples, as from *to_triples()*, into a Dmrs object.

to_dict(*short_pred=True*, *properties=True*)

Encode the Dmrs as a dictionary suitable for JSON serialization.

to_triples(*short_pred=True*, *properties=True*)

Encode the Dmrs as triples suitable for PENMAN serialization.

class delphin.mrs.xmrs.**Mrs**(*top=None*, *index=None*, *xarg=None*, *rels=None*, *hcons=None*, *icons=None*, *lnk=None*, *surface=None*, *identifier=None*, *vars=None*)

Construct an *Xmrs* using MRS components.

Formally, Minimal Recursion Semantics (MRS) have a top handle, a bag of Elementary Predications, and a bag of Handle Constraints. All arguments, including intrinsic arguments and constant arguments, are expected to be contained by the EPs.

Parameters

- **top** – the TOP (or LTOP) variable
- **index** – the INDEX variable
- **xarg** – the XARG variable
- **rels** – an iterable of ElementaryPredications
- **hcons** – an iterable of HandleConstraints

- **icons** – an iterable of IndividualConstraints
- **lnk** – the Lnk object associating the MRS to the surface form
- **surface** – the surface string
- **identifier** – a discourse-utterance id
- **vars** – a mapping of variables to a list of (property, value) pairs

Example:

```
>>> m = Mrs(
>>>     top='h0',
>>>     index='e2',
>>>     rels=[ElementaryPredication(
>>>         Pred.surface('_rain_v_1_rel'),
>>>         label='h1',
>>>         args={'ARG0': 'e2'},
>>>         vars={'e2': {'SF': 'prop-or-ques', 'TENSE': 'present'}}
>>>     )],
>>>     hcons=[HandleConstraint('h0', 'qeq', 'h1')]
>>> )
```

classmethod from_dict(*d*)

Decode a dictionary, as from `to_dict()`, into an Mrs object.

to_dict(*short_pred=True*, *properties=True*)

Encode the Mrs as a dictionary suitable for JSON serialization.

delphin.mrs.xmrs.**Rmrs**(*top=None*, *index=None*, *xarg=None*, *eps=None*, *args=None*, *hcons=None*, *icons=None*, *lnk=None*, *surface=None*, *identifier=None*, *vars=None*)

Construct an `Xmrs` from RMRS components.

Robust Minimal Recursion Semantics (RMRS) are like MRS, but all predications have a nodeid ("anchor"), and arguments are not contained by the source predications, but instead reference the nodeid of their predication.

Parameters

- **top** – the TOP (or maybe LTOP) variable
- **index** – the INDEX variable
- **xarg** – the XARG variable
- **eps** – an iterable of EPs
- **args** – a nested mapping of `{nodeid: {rargname: value}}`
- **hcons** – an iterable of HandleConstraint objects
- **icons** – an iterable of IndividualConstraint objects
- **lnk** – the Lnk object associating the MRS to the surface form
- **surface** – the surface string
- **identifier** – a discourse-utterance id
- **vars** – a mapping of variables to a list of `(property, value)` pairs

Example:

```
>>> m = Rmrs(
>>>     top='h0',
>>>     index='e2',
>>>     eps=[ElementaryPredication(
>>>         10000,
>>>         Pred.surface('_rain_v_1_rel'),
>>>         'h1'
>>>     )],
>>>     args={10000: {'ARG0': 'e2'}},
>>>     hcons=[HandleConstraint('h0', 'qeq', 'h1'),
>>>     vars={'e2': {'SF': 'prop-or-ques', 'TENSE': 'present'}}
>>> )
```

class delphin.mrs.xmrs.**Xmrs**(*top=None*, *index=None*, *xarg=None*, *eps=None*, *hcons=None*, *icons=None*, *vars=None*, *lnk=None*, *surface=None*, *identifier=None*)

Xmrs is a common class for Mrs, Rmrs, and Dmrs objects.

Parameters

- **top** – the TOP (or maybe LTOP) variable
- **index** – the INDEX variable
- **xarg** – the XARG variable
- **eps** – an iterable of EPs (see above)
- **hcons** – an iterable of HCONS (see above)
- **icons** – an iterable of ICONS (see above)
- **vars** – a mapping of variable to a list of property-value pairs
- **lnk** – the Lnk object associating the Xmrs to the surface form
- **surface** – the surface string
- **identifier** – a discourse-utterance id

Xmrs can be instantiated directly, but it may be more convenient to use the *Mrs()*, *Rmrs()*, or *Dmrs()* constructor functions.

Variables are simply strings, but must be of the proper form in order to be recognized as variables and not constants. The form is basically a sequence of non-integers followed by a sequence of integers, but see `delphin.mrs.components.var_re` for the regular expression used to determine a match.

The *eps* argument is an iterable of tuples representing ElementaryPredications. These can be objects of the ElementaryPredication class itself, or an equivalent tuple. The same goes for *hcons* and *icons* with the HandleConstraint and IndividualConstraint classes, respectively.

top
the top (i.e. LTOP) handle

index
the semantic index

xarg
the external argument

lnk
surface alignment

Type `Lnk`

surface
the surface string

identifier
a discourse-utterance ID (often unset)

add_eps(*eps*)
Incorporate the list of EPs given by *eps*.

add_hcons(*hcons*)
Incorporate the list of HandleConstraints given by *hcons*.

add_icons(*icons*)
Incorporate the individual constraints given by *icons*.

args(*nodeid*)
Return the arguments for the predication given by *nodeid*.

All arguments (including intrinsic and constant arguments) are included. MOD/EQ links are not considered arguments. If only arguments that target other predications are desired, see `outgoing_args()`.

Parameters **nodeid** – the nodeid of the EP that is the arguments' source

Returns *dict* – `{role: tgt}`

ep(*nodeid*)
Return the ElementaryPredication with the given *nodeid*.

eps(*nodeids=None*)
Return the EPs with the given *nodeid*, or all EPs.

Parameters **nodeids** – an iterable of nodeids of EPs to return; if `None`, return all EPs

classmethod from_xmrs(*xmrs*, ***kwargs*)
Facilitate conversion among subclasses.

Parameters

- **xmrs** (`Xmrs`) – instance to convert from; possibly an instance of a subclass, such as `Mrs` or `Dmrs`
- ****kwargs** – additional keyword arguments that may be used by a subclass's redefinition of `from_xmrs()`.

hcon(*hi*)
Return the HandleConstraint with high variable *hi*.

hcons()
Return the list of HCONS.

icons(*left=None*)
Return the ICONS with left variable *left*, or all ICONS.

Parameters **left** – the left variable of the ICONS to return; if `None`, return all ICONS

identifier = None
A discourse-utterance id

incoming_args(*nodeid*)
Return the arguments that target *nodeid*.

Valid arguments include regular variable arguments and scopal (label-selecting or HCONS) arguments. MOD/EQ links and intrinsic arguments are not included.

Parameters **nodeid** – the nodeid of the EP that is the arguments' target

Returns *dict* – `{source_nodeid: {rargname: value}}`

is_connected()

Return `True` if the Xmrs represents a connected graph.

Subgraphs can be connected through things like arguments, QEQs, and label equalities.

is_well_formed()

Return `True` if the Xmrs is well-formed, `False` otherwise.

See *validate()*

label(*nodeid*)

Return the label of the predication given by *nodeid*

labels(*nodeids=None*)

Return the list of labels for *nodeids*, or all labels.

Parameters **nodeids** – an iterable of nodeids for predications to get labels from; if `None`, return labels for all predications

---

Note: This returns the label of each predication, even if it's shared by another predication. Thus, `zip(nodeids, xmrs.labels(nodeids))` will pair nodeids with their labels.

Returns A list of labels

labelset(*label*)

Return the set of nodeids for predications that share *label*.

Parameters **label** – the label that returned nodeids share.

Returns A set of nodeids, which may be an empty set.

labelset_heads(*label*)

Return the heads of the labelset selected by *label*.

Parameters **label** – the label from which to find head nodes/EPs.

Returns An iterable of nodeids.

lnk = None

A Lnk object to associate the Xmrs to the surface form

ltop

The top handle if specified; `None` otherwise.

Note: Equivalent to *top*

---

nodeid(*iv*, *quantifier=False*)

Return the nodeid of the predication selected by *iv*.

Parameters

- **iv** – the intrinsic variable of the predication to select
- **quantifier** – if `True`, treat *iv* as a bound variable and find its quantifier; otherwise the non-quantifier will be returned

nodeids(*ivs=None*, *quantifier=None*)

Return the list of nodeids given by *ivs*, or all nodeids.

Parameters

- **ivs** – the intrinsic variables of the predications to select; if `None`, return all nodeids (but see *quantifier*)
- **quantifier** – if `True`, only return nodeids of quantifiers; if `False`, only return non-quantifiers; if `None` (the default), return both

outgoing_args(*nodeid*)

Return the arguments going from *nodeid* to other predications.

Valid arguments include regular variable arguments and scopal (label-selecting or HCONS) arguments. MOD/EQ links, intrinsic arguments, and constant arguments are not included.

Parameters **nodeid** – the nodeid of the EP that is the arguments' source

Returns *dict* – `{role: tgt}`

pred(*nodeid*)

Return the Pred object for the predications given by *nodeid*.

preds(*nodeids=None*)

Return the Pred objects for *nodeids*, or all Preds.

Parameters **nodeids** – an iterable of nodeids of predications to return Preds from; if `None`, return all Preds

properties(*var_or_nodeid*, *as_list=False*)

Return a dictionary of variable properties for *var_or_nodeid*.

Parameters **var_or_nodeid** – if a variable, return the properties associated with the variable; if a nodeid, return the properties associated with the intrinsic variable of the predication given by the nodeid

subgraph(*nodeids*)

Return an Xmrs object with only the specified *nodeids*.

Necessary variables and arguments are also included in order to connect any nodes that are connected in the original Xmrs.

Parameters **nodeids** – the nodeids of the nodes/EPs to include in the subgraph.

Returns An *Xmrs* object.

surface = None

The surface string

validate()

Check that the Xmrs is well-formed.

The Xmrs is analyzed and a list of problems is compiled. If any problems exist, an `XmrsError` is raised with the list joined as the error message. A well-formed Xmrs has the following properties:

- All predications have an intrinsic variable
- Every intrinsic variable belongs one predication and maybe one quantifier
- Every predication has no more than one quantifier
- All predications have a label
- The graph of predications form a net (i.e. are connected). Connectivity can be established with variable arguments, QEQs, or label-equality.
- The lo-handle for each QEQ must exist as the label of a predication

variables()

Return the list of all variables.

CHAPTER 13

delphin.repp

Regular Expression Preprocessor (REPP)

A Regular-Expression Preprocessor [REPP] is a method of applying a system of regular expressions for transformation and tokenization while retaining character indices from the original input string.

class `delphin.repp.`**`REPP`**(*name=None*, *modules=None*, *active=None*)

A Regular Expression Pre-Processor (REPP).

The normal way to create a new REPP is to read a .rpp file via the `from_file()` classmethod. For REPPs that are defined in code, there is the `from_string()` classmethod, which parses the same definitions but does not require file I/O. Both methods, as does the class's `__init__()` method, allow for pre-loaded and named external *modules* to be provided, which allow for external group calls (also see `from_file()` or implicit module loading). By default, all external submodules are deactivated, but they can be activated by adding the module names to *active* or, later, via the `activate()` method.

A third classmethod, `from_config()`, reads a PET-style configuration file (e.g., `repp.set`) which may specify the available and active modules, and therefore does not take the *modules* and *active* parameters.

Parameters

- **name** (*str, optional*) – the name assigned to this module
- **modules** (*dict, optional*) – a mapping from identifiers to REPP modules
- **active** (*iterable, optional*) – an iterable of default module activations

activate(*mod*)

Set external module *mod* to active.

apply(*s*, *active=None*)

Apply the REPP's rewrite rules to the input string *s*.

Parameters

- **s** (*str*) – the input string to process
- **active** (*optional*) – a collection of external module names that may be applied if called

Returns

a ***REPPResult*** **object containing the processed** string and characterization maps

deactivate(*mod*)
Set external module *mod* to inactive.

classmethod from_config(*path*, *directory=None*)
Instantiate a REPP from a PET-style `.set` configuration file.

The *path* parameter points to the configuration file. Submodules are loaded from *directory*. If *directory* is not given, it is the directory part of *path*.

Parameters

- **path** (*str*) – the path to the REPP configuration file
- **directory** (*str, optional*) – the directory in which to search for submodules

classmethod from_file(*path*, *directory=None*, *modules=None*, *active=None*)
Instantiate a REPP from a `.rpp` file.

The *path* parameter points to the top-level module. Submodules are loaded from *directory*. If *directory* is not given, it is the directory part of *path*.

A REPP module may utilize external submodules, which may be defined in two ways. The first method is to map a module name to an instantiated REPP instance in *modules*. The second method assumes that an external group call `>abc` corresponds to a file `abc.rpp` in *directory* and loads that file. The second method only happens if the name (e.g., `abc`) does not appear in *modules*. Only one module may define a tokenization pattern.

Parameters

- **path** (*str*) – the path to the base REPP file to load
- **directory** (*str, optional*) – the directory in which to search for submodules
- **modules** (*dict, optional*) – a mapping from identifiers to REPP modules
- **active** (*iterable, optional*) – an iterable of default module activations

classmethod from_string(*s*, *name=None*, *modules=None*, *active=None*)
Instantiate a REPP from a string.

Parameters

- **name** (*str, optional*) – the name of the REPP module
- **modules** (*dict, optional*) – a mapping from identifiers to REPP modules
- **active** (*iterable, optional*) – an iterable of default module activations

tokenize(*s*, *pattern=None*, *active=None*)
Rewrite and tokenize the input string *s*.

Parameters

- **s** (*str*) – the input string to process
- **pattern** (*str, optional*) – the regular expression pattern on which to split tokens; defaults to `[ ]+`
- **active** (*optional*) – a collection of external module names that may be applied if called

Returns a *YyTokenLattice* containing the tokens and their characterization information

trace(*s*, *active=None*, *verbose=False*)

Rewrite string *s* like `apply()`, but yield each rewrite step.

Parameters

- **s** (*str*) – the input string to process
- **active** (*optional*) – a collection of external module names that may be applied if called
- **verbose** (*bool, optional*) – if `False`, only output rules or groups that matched the input

Yields

a ***REPPStep*** **object for each intermediate rewrite** step, and finally a *REPPResult* object after the last rewrite

class delphin.repp.**REPPResult**

The final result of REPP application.

string

resulting string after all rules have applied

Type str

startmap

integer array of start offsets

Type `array`

endmap

integer array of end offsets

Type `array`

class delphin.repp.**REPPStep**

A single rule application in REPP.

input

input string (prior to application)

Type str

output

output string (after application)

Type str

operation

operation performed

applied

`True` if the rule was applied

Type bool

startmap

integer array of start offsets

Type `array`

endmap

integer array of end offsets

Type `array`

CHAPTER 14

delphin.tdl

Contents

Classes and functions for parsing and inspecting TDL.

This module makes it easy to inspect what is written on definitions in Type Description Language (TDL), but it doesn't interpret type hierarchies (such as by performing unification, subsumption calculations, or creating GLB types). That is, while it wouldn't be useful for creating a parser, it is useful if you want to statically inspect the types in a grammar and the constraints they apply.

TDL was originally described in Krieger and Schäfer, 1994 [KS1994], but it describes many features not in use by the DELPH-IN variant, such as disjunction. Copestake, 2002 [COP2002] better describes the subset in use by DELPH-IN, but it has become outdated and its TDL syntax description is inaccurate in places, but it is still a great resource for understanding the interpretation of TDL grammar descriptions. The TdlRfc page of the DELPH-IN Wiki contains the most up-to-date description of the TDL syntax used by DELPH-IN grammars, including features such as documentation strings and regular expressions.

14.1 Module Parameters

Some aspects of TDL parsing can be customized per grammar, and the following module variables may be reassigned to accommodate those differences. For instance, in the ERG, the type used for list feature structures is `*list*`, while

for Matrix-based grammars it is `list`. PyDelphin defaults to the values used by the ERG.

`delphin.tdl.`**`LIST_TYPE = '*list*'`**
type of lists in TDL

`delphin.tdl.`**`EMPTY_LIST_TYPE = '*null*'`**
type of list terminators

`delphin.tdl.`**`LIST_HEAD = 'FIRST'`**
feature for list items

`delphin.tdl.`**`LIST_TAIL = 'REST'`**
feature for list tails

`delphin.tdl.`**`DIFF_LIST_LIST = 'LIST'`**
feature for diff-list lists

`delphin.tdl.`**`DIFF_LIST_LAST = 'LAST'`**
feature for the last path in a diff-list

14.2 Functions

`delphin.tdl.`**`iterparse`**(*source*, *encoding='utf-8'*)
Parse the TDL file *source* and iteratively yield parse events.

If *source* is a filename, the file is opened and closed when the generator has finished, otherwise *source* is an open file object and will not be closed when the generator has finished.

Parse events are `(event, object, lineno)` tuples, where `event` is a string (`"TypeDefinition"`, `"TypeAddendum"`, `"LexicalRuleDefinition"`, `"LetterSet"`, `"WildCard"`, `"LineComment"`, or `"BlockComment"`), `object` is the interpreted TDL object, and `lineno` is the line number where the entity began in *source*.

Parameters

- **`source`** (*`str`, `file`*) – a filename or open file object
- **`encoding`** (*`str`*) – the encoding of the file (default: `"utf-8"`; ignored if *source* is an open file)

Yields `(event, object, lineno)` tuples

Example

```
>>> lex = {}
>>> for event, obj, lineno in tdl.iterparse('erg/lexicon.tdl'):
...     if event == 'TypeDefinition':
...         lex[obj.identifier] = obj
...
>>> lex['eucalyptus_n1']['SYNSEM.LKEYS.KEYREL.PRED']
<String object (_eucalyptus_n_1_rel) at 140625748595960>
```

`delphin.tdl.`**`format`**(*obj*, *indent=0*)
Serialize TDL objects to strings.

Parameters

- **`obj`** – instance of *`Term`*, *`Conjunction`*, or *`TypeDefinition`* classes or subclasses

- **indent** (*int*) – number of spaces to indent the formatted object

Returns *str* – serialized form of *obj*

Example

```
>>> conj = tdl.Conjunction([
...     tdl.TypeIdentifier('lex-item'),
...     tdl.AVM([('SYNSEM.LOCAL.CAT.HEAD.MOD',
...               tdl.ConsList(end=tdl.EMPTY_LIST_TYPE))])
... ])
>>> t = tdl.TypeDefinition('non-mod-lex-item', conj)
>>> print(format(t))
non-mod-lex-item := lex-item &
  [ SYNSEM.LOCAL.CAT.HEAD.MOD < > ].
```

14.3 Classes

The TDL entity classes are the objects returned by *iterparse()*, but they may also be used directly to build TDL structures, e.g., for serialization.

14.3.1 Terms

class delphin.tdl.**Term**(*docstring=None*)

Base class for the terms of a TDL conjunction.

All terms are defined to handle the binary '&' operator, which puts both into a Conjunction:

```
>>> TypeIdentifier('a') & TypeIdentifier('b')
<Conjunction object at 140008950372168>
```

Parameters **docstring** (*str*) – documentation string

docstring

documentation string

Type str

class delphin.tdl.**TypeTerm**(*string*, *docstring=None*)

Bases: *delphin.tdl.Term*, *str*

Base class for type terms (identifiers, strings and regexes).

This subclass of *Term* also inherits from *str* and forms the superclass of the string-based terms *TypeIdentifier*, *String*, and *Regex*. Its purpose is to handle the correct instantiation of both the *Term* and *str* supertypes and to define equality comparisons such that different kinds of type terms with the same string value are not considered equal:

```
>>> String('a') == String('a')
True
>>> String('a') == TypeIdentifier('a')
False
```

class delphin.tdl.**TypeIdentifier**(*string*, *docstring=None*)

Bases: *delphin.tdl.TypeTerm*

Type identifiers, or type names.

Unlike other *TypeTerms*, TypeIdentifiers use case-insensitive comparisons:

```
>>> TypeIdentifier('MY-TYPE') == TypeIdentifier('my-type')
True
```

Parameters

- **string** (*str*) – type name
- **docstring** (*str*) – documentation string

docstring

documentation string

Type str

class delphin.tdl.**String**(*string*, *docstring=None*)

Bases: *delphin.tdl.TypeTerm*

Double-quoted strings.

Parameters

- **string** (*str*) – type name
- **docstring** (*str*) – documentation string

docstring

documentation string

Type str

class delphin.tdl.**Regex**(*string*, *docstring=None*)

Bases: *delphin.tdl.TypeTerm*

Regular expression patterns.

Parameters

- **string** (*str*) – type name
- **docstring** (*str*) – documentation string

docstring

documentation string

Type str

class delphin.tdl.**AVM**(*featvals=None*, *docstring=None*)

Bases: *delphin.tfs.FeatureStructure*, *delphin.tdl.Term*

A feature structure as used in TDL.

Parameters

- **featvals** (*list, dict*) – a sequence of `(attribute, value)` pairs or an attribute to value mapping
- **docstring** (*str*) – documentation string

docstring
documentation string

Type str

features(*expand=False*)
Return the list of tuples of feature paths and feature values.

Parameters **expand** (*bool*) – if `True`, expand all feature paths

Example

```
>>> avm = AVM([('A.B', TypeIdentifier('1')),
...            ('A.C', TypeIdentifier('2'))])
>>> avm.features()
[('A', <AVM object at ...>)]
>>> avm.features(expand=True)
[('A.B', <TypeIdentifier object (1) at ...>),
 ('A.C', <TypeIdentifier object (2) at ...>)]
```

normalize()
Reduce trivial AVM conjunctions to just the AVM.

For example, in `[ ATTR1 [ ATTR2 val ] ]` the value of `ATTR1` could be a conjunction with the sub-AVM `[ ATTR2 val ]`. This method removes the conjunction so the sub-AVM nests directly (equivalent to `[ ATTR1.ATTR2 val ]` in TDL).

class delphin.tdl.**ConsList**(*values=None*, *end='*list*'*, *docstring=None*)
Bases: *delphin.tdl.AVM*

AVM subclass for cons-lists (`< ... >`)

This provides a more intuitive interface for creating and accessing the values of list structures in TDL. Some combinations of the *values* and *end* parameters correspond to various TDL forms as described in the table below:

TDL form	values	end	state
`< >`	`None`	`EMPTY_LIST_TYPE`	closed
`< ... >`	`None`	`LIST_TYPE`	open
`< a >`	`[a]`	`EMPTY_LIST_TYPE`	closed
`< a, b >`	`[a, b]`	`EMPTY_LIST_TYPE`	closed
`< a, ... >`	`[a]`	`LIST_TYPE`	open
`< a . b >`	`[a]`	`b`	closed

Parameters

- **values** (*list*) – a sequence of *Conjunction* or *Term* objects to be placed in the AVM of the list.
- **end** (str, *Conjunction*, *Term*) – last item in the list (default: *LIST_TYPE*) which determines if the list is open or closed
- **docstring** (*str*) – documentation string

terminated
if `False`, the list can be further extended by following the *LIST_TAIL* features.

Type bool

docstring
documentation string

Type str

append(*value*)
Append an item to the end of an open ConsList.

Parameters **value** (*Conjunction*, *Term*) – item to add

Raises TdlError – when appending to a closed list

terminate(*end*)
Set the value of the tail of the list.

Adding values via *append()* places them on the FIRST feature of some level of the feature structure (e.g., REST.FIRST), while *terminate()* places them on the final REST feature (e.g., REST.REST). If *end* is a *Conjunction* or *Term*, it is typically a *Coreference*, otherwise *end* is set to tdl.EMPTY_LIST_TYPE or tdl.LIST_TYPE. This method does not necessarily close the list; if *end* is tdl.LIST_TYPE, the list is left open, otherwise it is closed.

Parameters

- **end** (str, *Conjunction*, *Term*) – value to
- **as the end of the list.** (*use*) –

values()
Return the list of values in the ConsList feature structure.

class delphin.tdl.**DiffList**(*values=None*, *docstring=None*)
Bases: *delphin.tdl.AVM*

AVM subclass for diff-lists (<! ... !>)

As with *ConsList*, this provides a more intuitive interface for creating and accessing the values of list structures in TDL. Unlike *ConsList*, DiffLists are always closed lists with the last item coreferenced with the LAST feature, which allows for the joining of two diff-lists.

Parameters

- **values** (*list*) – a sequence of *Conjunction* or *Term* objects to be placed in the AVM of the list
- **docstring** (*str*) – documentation string

last
the feature path to the list position coreferenced by the value of the *DIFF_LIST_LAST* feature.

Type str

docstring
documentation string

Type str

values()
Return the list of values in the DiffList feature structure.

class delphin.tdl.**Coreference**(*identifier*, *docstring=None*)
Bases: *delphin.tdl.Term*

TDL coreferences, which represent re-entrancies in AVMs.

Parameters

- **identifier** (*str*) – identifier or tag associated with the coreference; for internal use (e.g., in *DiffList* objects), the identifier may be None
- **docstring** (*str*) – documentation string

identifier
corefernce identifier or tag

Type str

docstring
documentation string

Type str

14.3.2 Conjunctions

class delphin.tdl.**Conjunction**(*terms=None*)
Conjunction of TDL terms.

Parameters **terms** (*list*) – sequence of *Term* objects

add(*term*)
Add a term to the conjunction.

Parameters **term** (*Term*, *Conjunction*) – term to add; if a *Conjunction*, all of its terms are added to the current conjunction.

Raises TypeError – when *term* is an invalid type

features(*expand=False*)
Return the list of feature-value pairs in the conjunction.

get(*key*, *default=None*)
Get the value of attribute *key* in any AVM in the conjunction.

Parameters

- **key** – attribute path to search
- **default** – value to return if *key* is not defined on any AVM

normalize()
Rearrange the conjunction to a conventional form.

This puts any coreference(s) first, followed by type terms, then followed by AVM(s) (including lists). AVMs are normalized via *AVM.normalize()*.

string()
Return the first string term in the conjunction, or None.

terms
The list of terms in the conjunction.

types()
Return the list of type terms in the conjunction.

14.3.3 Type and Instance Definitions

class delphin.tdl.**TypeDefinition**(*identifier*, *conjunction*, *docstring=None*)
A top-level Conjunction with an identifier.

Parameters

- **identifier** (*str*) – type name
- **conjunction** (*Conjunction*, *Term*) – type constraints
- **docstring** (*str*) – documentation string

identifier
type identifier

Type str

conjunction
type constraints

Type *Conjunction*

docstring
documentation string

Type str

documentation(*level='first'*)
Return the documentation of the type.

By default, this is the first docstring on a top-level term. By setting *level* to "top", the list of all docstrings on top-level terms is returned, including the type's `docstring` value, if not `None`, as the last item. The docstring for the type itself is available via *TypeDefinition.docstring*.

Parameters **level** (*str*) – "first" or "top"

Returns a single docstring or a list of docstrings

features(*expand=False*)
Return the list of feature-value pairs in the conjunction.

supertypes
The list of supertypes for the type.

class delphin.tdl.**TypeAddendum**(*identifier*, *conjunction=None*, *docstring=None*)
Bases: *delphin.tdl.TypeDefinition*

An addendum to an existing type definition.

Type addenda, unlike *type definitions*, do not require supertypes, or even any feature constraints. An addendum, however, must have at least one supertype, AVM, or docstring.

Parameters

- **identifier** (*str*) – type name
- **conjunction** (*Conjunction*, *Term*) – type constraints
- **docstring** (*str*) – documentation string

identifier
type identifier

Type str

conjunction
type constraints

Type *Conjunction*

docstring
documentation string

Type str

class delphin.tdl.**LexicalRuleDefinition**(*identifier*, *affix_type*, *patterns*, *conjunction*, ***kwargs*)

Bases: *delphin.tdl.TypeDefinition*

An inflecting lexical rule definition.

Parameters

- **identifier** (*str*) – type name
- **affix_type** (*str*) – "prefix" or "suffix"
- **patterns** (*list*) – sequence of (match, replacement) pairs
- **conjunction** (*Conjunction*, *Term*) – conjunction of constraints applied by the rule
- **docstring** (*str*) – documentation string

identifier
type identifier

Type str

affix_type
"prefix" or "suffix"

Type str

patterns
sequence of (match, replacement) pairs

Type list

conjunction
type constraints

Type *Conjunction*

docstring
documentation string

Type str

14.3.4 Morphological Patterns

class delphin.tdl.**LetterSet**(*var*, *characters*)

A capturing character class for inflectional lexical rules.

LetterSets define a pattern (e.g., "!a") that may match any one of its associated characters. Unlike *WildCard* patterns, LetterSet variables also appear in the replacement pattern of an affixing rule, where they insert the character matched by the corresponding letter set.

Parameters

- **var** (*str*) – variable used in affixing rules (e.g., "!a")
- **characters** (*str*) – string or collection of characters that may match an input character

var
letter-set variable

Type str

characters
characters included in the letter-set

Type str

class delphin.tdl.**WildCard**(*var*, *characters*)
A non-capturing character class for inflectional lexical rules.

WildCards define a pattern (e.g., "?a") that may match any one of its associated characters. Unlike *LetterSet* patterns, WildCard variables may not appear in the replacement pattern of an affixing rule.

Parameters

- **var** (*str*) – variable used in affixing rules (e.g., "!a")
- **characters** (*str*) – string or collection of characters that may match an input character

var
wild-card variable

Type str

characters
characters included in the wild-card

Type str

14.4 Deprecated

Use of the following functions are classes is no longer recommended, and they will be removed in a future version.

delphin.tdl.**parse**(*f*, *encoding='utf-8'*)
Parse the TDL file *f* and yield the interpreted contents.

If *f* is a filename, the file is opened and closed when the generator has finished, otherwise *f* is an open file object and will not be closed when the generator has finished.

Parameters

- **f** (*str, file*) – a filename or open file object
- **encoding** (*str*) – the encoding of the file (default: "utf-8"; ignored if *f* is an open file)

delphin.tdl.**lex**(*stream*)

delphin.tdl.**tokenize**(*s*)
Tokenize a string *s* of TDL code.

class delphin.tdl.**TdlDefinition**(*supertypes=None*, *featvals=None*)
Bases: *delphin.tfs.FeatureStructure*

A typed feature structure with supertypes.

A TdlDefinition is like a *FeatureStructure* but each structure may have a list of supertypes.

local_constraints()
Return the constraints defined in the local AVM.

class delphin.tdl.**TdlConsList**(*supertypes=None*, *featvals=None*)
Bases: *delphin.tdl.TdlDefinition*

A TdlDefinition for cons-lists (< ... >)

Navigating the feature structure for lists can be cumbersome, so this subclass of `TdlDefinition` provides the `values()` method to collect the items nested inside the list and return them as a Python list.

values()
Return the list of values.

class delphin.tdl.**TdlDiffList**(*supertypes=None*, *featvals=None*)
Bases: `delphin.tdl.TdlDefinition`

A TdlDefinition for diff-lists (<! ... !>)

Navigating the feature structure for lists can be cumbersome, so this subclass of `TdlDefinition` provides the `values()` method to collect the items nested inside the list and return them as a Python list.

values()
Return the list of values.

class delphin.tdl.**TdlType**(*identifier*, *definition*, *coreferences=None*, *docstring=None*)
Bases: `delphin.tdl.TdlDefinition`

A top-level TdlDefinition with an identifier.

Parameters

- **identifier** (`str`) – type name
- **definition** (`TdlDefinition`) – definition of the type
- **coreferences** (`list`) – (tag, paths) tuple of coreferences, where paths is a list of feature paths that share the tag
- **docstring** (`list`) – list of documentation strings

class delphin.tdl.**TdlInflRule**(*identifier*, *affix=None*, ***kwargs*)
Bases: `delphin.tdl.TdlType`

TDL inflectional rule.

Parameters

- **identifier** (`str`) – type name
- **affix** (`str`) – inflectional affixes

CHAPTER 15

delphin.tfs

Basic classes for modeling feature structures.

This module defines the `FeatureStructure` and `TypedFeatureStructure` classes, which model an attribute value matrix (AVM), with the latter including an associated type. They allow feature access through TDL-style dot notation regular dictionary keys.

In addition, the `TypeHierarchy` class implements a multiple-inheritance hierarchy with checks for type subsumption and compatibility.

class delphin.tfs.**FeatureStructure**(*featvals=None*)

A feature structure.

This class manages the access of nested features using dot-delimited notation (e.g., `SYNSEM.LOCAL.CAT.HEAD`).

Parameters **featvals** (*dict, list*) – a mapping or iterable of feature paths to feature values

features(*expand=False*)

Return the list of tuples of feature paths and feature values.

Parameters **expand** (*bool*) – if `True`, expand all feature paths

Example

```
>>> fs = FeatureStructure([('A.B', 1), ('A.C', 2)])
>>> fs.features()
[('A', <FeatureStructure object at ...>)]
>>> fs.features(expand=True)
[('A.B', 1), ('A.C', 2)]
```

get(*key, default=None*)

Return the value for *key* if it exists, otherwise *default*.

class delphin.tfs.**TypeHierarchy**(*top, hierarchy=None*)

A Type Hierarchy.

Type hierarchies have certain properties, such as a unique top node, multiple inheritance, and unique greatest-lower-bound (glb) types.

Note: Checks for unique glbs is not yet implemented.

Parameters

- **top** (*str*) – unique top type
- **hierarchy** (*dict*) – mapping of `{child: [parents]}`

ancestors(*typename*)
Return the ancestor types of *typename*.

compatible(*a*, *b*)
Return `True` if type *a* is compatible with type *b*.

descendants(*typename*)
Return the descendant types of *typename*.

subsumes(*a*, *b*)
Return `True` if type *a* subsumes type *b*.

class delphin.tfs.**TypedFeatureStructure**(*type*, *featvals=None*)
A typed *FeatureStructure*.

Parameters

- **type** (*str*) – type name
- **featvals** (*dict, list*) – a mapping or iterable of feature paths to feature values

CHAPTER 16

delphin.tokens

class delphin.tokens.**YyToken**

A tuple of token data in the YY format.

Parameters

- **id** – token identifier
- **start** – start vertex
- **end** – end vertex
- **lnk** – <from:to> charspan (optional)
- **paths** – path membership
- **form** – surface token
- **surface** – original token (optional; only if `form` was modified)
- **ipos** – length of `lrules`? always 0?
- **lrules** – something about lexical rules; always "null"?
- **pos** – pairs of (POS, prob)

classmethod from_dict(*d*)

Decode from a dictionary as from `to_dict()`.

to_dict()

Encode the token as a dictionary suitable for JSON serialization.

class delphin.tokens.**YyTokenLattice**(*tokens*)

A lattice of YY Tokens.

Parameters **tokens** – a list of YyToken objects

classmethod from_list(*toks*)

Decode from a list as from `to_list()`.

classmethod from_string(*s*)

Decode from the YY token lattice format.

to_list()

Encode the token lattice as a list suitable for JSON serialization.

CHAPTER 17

delphin.tsql

See also:

The *select* command is a quick way to query test suites with TSQL queries.

TSQL – Test Suite Query Language

This module implements a subset of TSQL, namely the 'select' (or 'retrieve') queries for extracting data from test suites. The general form of a select query is:

```
[select] <projection> [from <tables>] [where <condition>]*
```

For example, the following selects item identifiers that took more than half a second to parse:

```
select i-id from item where total > 500
```

The `select` string is necessary when querying with the generic *query()* function, but is implied and thus disallowed when using the *select()* function.

The `<projection>` is a list of space-separated field names (e.g., `i-id i-input mrs`), or the special string `*` which selects all columns from the joined tables.

The optional `from` clause provides a list of table names (e.g., `item parse result`) that are joined on shared keys. The `from` clause is required when `*` is used for the projection, but it can also be used to select columns from non-standard tables (e.g., `i-id from output`). Alternatively, `delphin.itsdb`-style data specifiers (see *delphin.itsdb.get_data_specifier()*) may be used to specify the table on the column name (e.g., `item:i-id`).

The `where` clause provide conditions for filtering the list of results. Conditions are binary operations that take a column or data specifier on the left side and an integer (e.g., `10`), a date (e.g., `2018-10-07`), or a string (e.g., `"sleep"`) on the right side of the operator. The allowed conditions are:

Condition	Form
Regex match	`<field> ~ "regex"`
Regex fail	`<field> !~ "regex"`
Equality	`<field> = (integer|date|"string")`
Inequality	`<field> != (integer|date|"string")`
Less-than	`<field> < (integer|date)`
Less-or-equal	`<field> <= (integer|date)`
Greater-than	`<field> > (integer|date)`
Greater-or-equal	`<field> >= (integer|date)`

Boolean operators can be used to join multiple conditions or for negation:

Operation	Form
Disjunction	`X | Y`, `X || Y`, or `X or Y`
Conjunction	`X & Y`, `X && Y`, or `X and Y`
Negation	`!X` or `not X`

Normally, disjunction scopes over conjunction, but parentheses may be used to group clauses, so the following are equivalent:

```
... where i-id = 10 or i-id = 20 and i-input ~ "[Dd]og"
... where i-id = 10 or (i-id = 20 and i-input ~ "[Dd]og")
```

Multiple `where` clauses may also be used as a conjunction that scopes over disjunction, so the following are equivalent:

```
... where (i-id = 10 or i-id = 20) and i-input ~ "[Dd]og"
... where i-id = 10 or i-id = 20 where i-input ~ "[Dd]og"
```

This facilitates query construction, where a user may want to apply additional global constraints by appending new conditions to the query string.

PyDelphin has several differences to standard TSQL:

- `select *` requires a `from` clause
- `select * from item result` does not also include columns from the intervening `parse` table
- `select i-input from result` returns a matching `i-input` for every row in `result`, rather than only the unique rows

PyDelphin also adds some features to standard TSQL:

- optional table specifications on columns (e.g., `item:i-id`)
- multiple `where` clauses (as described above)

`delphin.tsql.`**`inspect_query`**(*query*)

Parse *query* and return the interpreted query object.

Example

```
>>> from delphin import tsql
>>> from pprint import pprint
>>> pprint(tsql.inspect_query(
```

(continues on next page)

(continued from previous page)

```
...     'select i-input from item where i-id < 100'))
{'querytype': 'select',
 'projection': ['i-input'],
 'tables': ['item'],
 'where': ('<', ('i-id', 100))}
```

`delphin.tsql.query`(*query*, *ts*, ***kwargs*)

Perform *query* on the testsuite *ts*.

Note: currently only 'select' queries are supported.

Parameters

- **query** (*str*) – TSQL query string
- **ts** (*delphin.itsdb.TestSuite*) – testsuite to query over
- **kwargs** – keyword arguments passed to the more specific query function (e.g., *select()*)

Example

```
>>> list(tsql.query('select i-id where i-length < 4', ts))
[[142], [1061]]
```

`delphin.tsql.select`(*query*, *ts*, *mode='list'*, *cast=True*)

Perform the TSQL selection query *query* on testsuite *ts*.

Note: The `select/retrieve` part of the query is not included.

Parameters

- **query** (*str*) – TSQL select query
- **ts** (*delphin.itsdb.TestSuite*) – testsuite to query over
- **mode** (*str*) – how to return the results (see *delphin.itsdb.select_rows()* for more information about the *mode* parameter; default: `list`)
- **cast** (*bool*) – if `True`, values will be cast to their datatype according to the testsuite's relations (default: `True`)

Example

```
>>> list(tsql.select('i-id where i-length < 4', ts))
[[142], [1061]]
```

CHAPTER 18

Indices and tables

- genindex
- modindex
- search

Bibliography

[MRS] Copestake, Ann, Dan Flickinger, Carl Pollard, and Ivan A. Sag. "Minimal recursion semantics: An introduction." Research on language and computation 3, no. 2-3 (2005): 281-332.

[RMRS] Copestake, Ann. "Report on the design of RMRS." DeepThought project deliverable (2003).

[EDS] Stephan Oepen, Dan Flickinger, Kristina Toutanova, and Christopher D Manning. Lingo Redwoods. Research on Language and Computation, 2(4):575–596, 2004.;

Stephan Oepen and Jan Tore Lønning. Discriminant-based MRS banking. In Proceedings of the 5th International Conference on Language Resources and Evaluation, pages 1250–1255, 2006.

[DMRS] Copestake, Ann. Slacker Semantics: Why superficiality, dependency and avoidance of commitment can be the right way to go. In Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics, pages 1–9. Association for Computational Linguistics, 2009.

[REPP] Rebecca Dridan and Stephan Oepen. Tokenization: Returning to a long solved problem—a survey, contrastive experiment, recommendations, and toolkit. In Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers), pages 378–382, Jeju Island, Korea, July 2012. Association for Computational Linguistics. URL http://www.aclweb.org/anthology/P12-2074.

[KS1994] Hans-Ulrich Krieger and Ulrich Schäfer. TDL: a type description language for constraint-based grammars. In Proceedings of the 15th conference on Computational linguistics, volume 2, pages 893–899. Association for Computational Linguistics, 1994.

[COP2002] Ann Copestake. Implementing typed feature structure grammars, volume 110. CSLI publications Stanford, 2002.

Python Module Index

d

Index

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y