PyDelphin Documentation

Release 1.3.0

Michael Wayne Goodman

Jul 01, 2020

GUIDES:

Quick Links

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

CHAPTER

ONE

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 3.6 and higher, regardless of the platform. Certain features, however, may require additional dependencies or may be platform specific, as shown in the table below:

Module or Function	Dependencies	Notes
`delphin.ace`	ACE	Linux and Mac only
`delphin.web.client`	requests	`[web]` extra
`delphin.web.server`	Falcon	`[web]` extra
`delphin.codecs.dmrspenman`	Penman	
`delphin.codecs.edspenman`	Penman	
`delphin.repp`	regex	`[repp]` extra

See *Installing Extra Dependencies* for information about installing with "extras", including those needed for PyDelphin development (which are not listed in the table above).

1.2 Installing from PyPI

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

```
[~]$ pip install pydelphin
```

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

Note: It is strongly recommended 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 Installing from Source

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

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

Install from the source code using **pip** as before but give it the path to the repository instead of the name of the PyPI project:

```
[~]$ cd pydelphin/
[~/pydelphin]$ pip install .
```

Installing from source does not require internet access once the repository has been cloned, but it does require internet to install any dependencies. Also note that if the project directory is named `pydelphin` (the default) and you install from the directory above it, you mustn't just use the directory name as this will cause **pip** to install from PyPI; instead, make it look path-like by prefixing it with `./` (i.e., `pip install ./pydelphin`).

For development, you may also want to use **pip**'s `-e` option to install PyDelphin as "editable", meaning it installs the dependencies but uses the local source files for PyDelphin's code, otherwise changes you make to PyDelphin won't be reflected in your (virtual) environment unless you reinstall PyDelphin.

> **Warning:** It is not recommended to install from source using `$ setup.py install`, because uninstalling or updating PyDelphin and its dependencies becomes more difficult.

1.4 Installing Extra Dependencies

Some features require dependencies beyond what the standard install provides. The purpose of keeping these dependencies optional is to reduce the install size for users who do not make use of the additional features.

If you need to use some of these features, such as `delphin.web` and `delphin.repp`, install the extra dependencies with **pip** as before but with an install parameter in brackets after `pydelphin`. For instance:

```
[~]$ pip install pydelphin[web,repp]
```

Without the install parameter, the PyDelphin code will still be installed but its dependencies will not be. The rest of PyDelphin will work but those features may raise an `ImportError` or issue a warning.

In addition, there are some dependencies that are only necessary for developers of PyDelphin to run unit tests and build documentation.

The extras that PyDelphin defines are as follows:

Extra	Description
`[web]`	Required for using the *`delphin.web`* client and server
`[repp]`	Optional for advanced regex features with *`delphin.repp`*
`[docs]`	Required for building documentation
`[tests]`	Required for running tests
`[dev]`	Required for making releases (includes `tests` and `docs`)

1.5 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, which is installed if you used the `test` or `dev` install parameters described above. Once **pytest** is installed (note: it may also be called **py.test**), run it to perform the unit tests:

```
[~/pydelphin]$ pytest
```

This will detect and run any unit tests it finds. It is best to run the **pytest** in a virtual environment with a clean install of PyDelphin to ensure that the local Python environment is not conflicting with PyDelphin's dependencies and also to ensure that PyDelphin specifies all its dependencies.

If you find it inconvenient to activate several virtual environments to test the supported Python versions, you may find **tox** useful. See tox's website for more information.

CHAPTER

TWO

WALKTHROUGH OF PYDELPHIN FEATURES

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

2.1 ACE and Web 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.web import client
>>> response = client.parse('Abrams chased Browne', params={'mrs': 'json'})
>>> response.result(0).mrs()
<MRS object (proper_q named chase_v_1 proper_q 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 Web API: http://moin.delph-in.net/ErgApi
- Bottlenose server: https://github.com/delph-in/bottlenose
- *`delphin.web`* module
- *`delphin.interface`* 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 import ace
>>> grm = '~/grammars/erg-2018-x86-64-0.9.30.dat'
>>> response = ace.parse(grm, 'Abrams chased Browne')
NOTE: parsed 1 / 1 sentences, avg 2135k, time 0.01316s
>>> response.result(0).mrs()
<MRS object (proper_q named chase_v_1 proper_q named) at 139897048034552>
```

See also:

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

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

2.2 Inspecting Semantic Structures

The original motivation for PyDelphin and the area with the most work is in modeling DELPH-IN Semantics representations such as MRS.

```
>>> m = response.result(0).mrs()
>>> [ep.predicate for ep in m.rels]
['proper_q', 'named', '_chase_v_1', 'proper_q', 'named']
>>> list(m.variables)
['h0', 'e2', 'h4', 'x3', 'h5', 'h6', 'h7', 'h1', 'x9', 'h10', 'h11', 'h12', 'h13']
>>> # get an EP by its ID (generally its intrinsic variable)
>>> m['x3']
<EP object (h7:named(CARG Abrams, ARG0 x3)) at 140709661206856>
>>> # quantifier IDs generally just replace 'x' with 'q'
>>> m['q3']
<EP object (h4:proper_q(ARG0 x3, RSTR h5, BODY h6)) at 140709661206760>
>>> # but if you want to be more careful you can do this...
>>> qmap = {p.iv: q for p, q in m.quantification_pairs()}
>>> qmap['x3']
<EP object (h4:proper_q(ARG0 x3, RSTR h5, BODY h6)) at 140709661206760>
>>> # EP arguments are available on the EPs
>>> m['e2'].args
{'ARG0': 'e2', 'ARG1': 'x3', 'ARG2': 'x9'}
>>> # While HCONS are available on the MRS
>>> [(hc.hi, hc.relation, hc.lo) for hc in m.hcons]
[('h0', 'qeq', 'h1'), ('h5', 'qeq', 'h7'), ('h11', 'qeq', 'h13')]
```

See also:

- Wiki of MRS topics: http://moin.delph-in.net/RmrsTop
- *`delphin.mrs`* module
- *Working with Semantic Structures*

Beyond the basic modeling of semantic structures, there are some semantic operations defined in the *`delphin.mrs`* module.

```
>>> from delphin import mrs
>>> mrs.is_isomorphic(m, m)
True
>>> mrs.is_isomorphic(m, response.result(1).mrs())
False
>>> mrs.has_intrinsic_variable_property(m)
True
>>> mrs.is_connected(m)
True
```

See also:

- MRS isomorphism wiki: http://moin.delph-in.net/MrsIsomorphism

Scoping semantic structures such as MRS and DMRS can make use of the *`delphin.scope`* module, which allows for inspection of the scope structures:

```
>>> from delphin import scope
>>> _response = ace.parse(grm, "Kim didn't think that Sandy left.")
>>> descendants = scope.descendants(_response.result(0).mrs())
>>> for id, ds in descendants.items():
...     print(m[id].predicate, [d.predicate for d in ds])
...
proper_q ['named']
named []
neg ['_think_v_1', '_leave_v_1']
_think_v_1 ['_leave_v_1']
_leave_v_1 []
proper_q ['named']
named []
```

See also:

- *delphin.scope* module

2.3 Converting Semantic Representations

Conversions between MRS, DMRS, and EDS representations are a single function call in PyDelphin. The converted representation has its own data structures so it can be inspected and manipulated in a natural way for the respective formalism. Here is DMRS conversion from MRS:

```
>>> from delphin import dmrs
>>> dmrs.from_mrs(m)
<DMRS object (proper_q named _chase_v_1 proper_q named) at 140709655360704>
```

And EDS conversion from MRS:

```
>>> from delphin import eds
>>> eds.from_mrs(m)
<EDS object (proper_q named _chase_v_1 proper_q named) at 140709655349560>
```

It is also possible to convert to MRS from DMRS.

2.4 Serializing Semantic Representations

The DELPH-IN community has designed many serialization formats of the semantic representations for various uses. For instance, the JSON formats are used in the Web API, and the PENMAN formats are sometimes used in machine learning applications. PyDelphin implements almost all of these formats, available in the *delphin.codecs* namespace.

```
>>> from delphin.codecs import simplemrs, mrx
>>> print(simplemrs.encode(m, indent=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" ] >
```

(continues on next page)

(continued from previous page)

```
  HCONS: < h0 qeq h1 h5 qeq h7 h11 qeq h13 > ]
>>> print(mrx.encode(m, indent=True))
<mrs cfrom="-1" cto="-1"><label vid="0" /><var sort="e" vid="2">
[...]
</mrs>
```

To serialize a different representation you must convert it first:

```
>>> d = dmrs.from_mrs(m)
>>> from delphin.codecs import dmrx
>>> print(dmrx.encode(d, indent=True))
<dmrs cfrom="-1" cto="-1" index="10002">
[...]
</dmrs>
>>> e = eds.from_mrs(m)
>>> from delphin.codecs import eds as edsnative  # avoid name collision
>>> print(edsnative.encode(e, indent=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:

- Wiki of MRS formats: http://moin.delph-in.net/MrsRfc
- *delphin.codecs* namespace

Some formats are currently export-only:

```
>>> from delphin.codecs import mrsprolog
>>> print(mrsprolog.encode(m, indent=True))
psoa(h0,e2,
  [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)]))
```

2.5 Tokens and Token Lattices

The Response object from the interface can return both the initial (string-level tokenization) and internal (token-mapped) tokens:

```
>>> 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

2.6 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

2.7 [incr tsdb()] TestSuites

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

```
>>> from delphin import itsdb
>>> ts = itsdb.TestSuite('~/grammars/erg/tsdb/gold/mrs')
>>> len(ts['item'])
107
>>> ts['item'][0]['i-input']
'It rained.'
>>> # modify a test suite in-memory
>>> ts['item'].update(0, {'i-input': 'It snowed.'})
```

(continues on next page)

(continued from previous page)

```
>>> ts['item'][0]['i-input']
'It snowed.'
>>> # TestSuite.commit() writes changes to disk
>>> ts.commit()
>>> # TestSuites can be parsed with a processor like ACE
>>> from delphin import ace
>>> with ace.ACEParser('~/grammars/erg-2018-x86-64-0.9.30.dat') as cpu:
...     ts.process(cpu)
...
NOTE: parsed 107 / 107 sentences, avg 4744k, time 2.93924s
```

See also:

- [incr tsdb()] wiki: http://moin.delph-in.net/ItsdbTop
- `delphin.itsdb` module
- `delphin.tsdb` module, for a low-level API
- *Working with [incr tsdb()] Test Suites*

2.8 TSQL Queries

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

```
>>> from delphin import tsql
>>> selection = tsql.select('i-id i-input where i-length > 5 && readings > 0', ts)
>>> next(iter(selection))
(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

2.9 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."):
```

(continues on next page)

(continued from previous page)

```
...     if isinstance(step, repp.REPPStep):
...         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

2.10 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('~/grammars/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()
```

(continues on next page)

(continued from previous page)

```
[('ORTH', <ConsList object at 140226925534472>), ('SYNSEM', <AVM object at␣
↪140226925299464>)]
>>> 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

2.11 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 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', 'RSTR', 'BODY', 'CARG']
>>> s.roles['ARG2']
'u'
>>> list(s.properties)
['bool', 'tense', 'mood', 'gender', 'number', 'person', 'pt', 'sf', '+', '-', 'tensed
↪', 'untensed', 'subjunctive', 'indicative', 'm-or-f', 'n', 'sg', 'pl', '1', '2', '3
↪', 'refl', 'std', 'zero', 'prop-or-ques', 'comm', 'past', 'pres', 'fut', 'm', 'f',
↪'prop', 'ques']
>>> s.properties.children('tense')
{'untensed', 'tensed'}
>>> s.properties.descendants('tense')
{'past', 'untensed', 'tensed', 'fut', 'pres'}
>>> len(s.predicates)
23403
>>> s.predicates['_cactus_n_1']
[Synopsis([SynopsisRole(ARG0, x, {'IND': '+'}, False)])]
>>> s.predicates.descendants('some_q')
{'_what+a_q', '_some_q_indiv', '_an+additional_q', '_another_q', '_many+a_q', '_a_q',
↪'_some_q', '_such+a_q'}
```

See also:

- The SEM-I wikis:
  - http://moin.delph-in.net/SemiRfc
  - http://moin.delph-in.net/RmrsSemi
- `delphin.semi` module

CHAPTER THREE

WORKING WITH SEMANTIC STRUCTURES

PyDelphin accommodates three kinds of semantic structures:

- `delphin.mrs` – Minimal Recursion Semantics
- `delphin.eds` – Elementary Dependency Structures
- `delphin.dmrs` – Dependency Minimal Recusion Semantics

MRS is the original underspecified representation in DELPH-IN, and is the only one directly output when parsing with DELPH-IN grammars. In PyDelphin, all three implement the `SemanticStructure` interface, while MRS and DMRS additionally implement the `ScopingSemanticStructure` interface. Common properties of `SemanticStructure` include a notion of the top of the graph and a list of `Predications`. The following ASCII-diagram illustrates the class hierarchy of these representations:

```
+-----------------------+
| delphin.lnk.LnkMixin |--------------------------+
+-----------------------+                          |
  |                                                |
  |  +------------------------------------+        |  +------------------------------+
  +--| delphin.sembase.SemanticStructure |        +--| delphin.sembase.Predication |
     +------------------------------------+           +------------------------------+
       |                                                |
       |  +------------------+                          |  +-------------------+
       +--| delphin.eds.EDS |                          +--| delphin.eds.Node |
       |  +------------------+                          |  +-------------------+
       |                                                |
       |  +-----------------------------------------+   |
       +--| delphin.scope.ScopingSemanticStructure |   |
          +-----------------------------------------+   |
            |                                           |
            |  +------------------+                     |  +-----------------+
            +--| delphin.mrs.MRS |                     +--| delphin.mrs.EP |
            |  +------------------+                     |  +-----------------+
            |                                           |
            |  +--------------------+                   |  +--------------------+
            +--| delphin.dmrs.DMRS |                   +--| delphin.dmrs.Node |
               +--------------------+                      +--------------------+
```

3.1 Basic Semantic Structures

The basic `SemanticStructure` interface provides methods for inspecting a structure's predications and arguments, morphosemantic properties, and quantification structure. First let's load an MRS to play with:

```
>>> from delphin.codecs import simplemrs
>>> # Load MRS for "They have enough capital to build a second factory."
>>> # (Tanaka Corpus i-id=30000034)
>>> m = simplemrs.decode('''
...   [ LTOP: h0 INDEX: e2 [ e SF: prop TENSE: pres MOOD: indicative PROG: - PERF: - ]
...     RELS: < [ pron<0:4> LBL: h4 ARG0: x3 [ x PERS: 3 NUM: pl IND: + PT: std ] ]
...             [ pronoun_q<0:4> LBL: h5 ARG0: x3 RSTR: h6 BODY: h7 ]
...             [ _have_v_1<5:9> LBL: h1 ARG0: e2 ARG1: x3 ARG2: x8 [ x PERS: 3 NUM:␣
↪sg ] ]
...             [ _enough_q<10:16> LBL: h9 ARG0: x8 RSTR: h10 BODY: h11 ]
...             [ _capital_n_1<17:24> LBL: h12 ARG0: x8 ]
...             [ with_p<25:51> LBL: h12 ARG0: e13 [ e SF: prop ] ARG1: e14 [ e SF:␣
↪prop-or-ques TENSE: untensed MOOD: indicative PROG: - PERF: - ] ARG2: x8 ]
...             [ _build_v_1<28:33> LBL: h12 ARG0: e14 ARG1: i15 ARG2: x16 [ x PERS:␣
↪3 NUM: sg IND: + ] ]
...             [ _a_q<34:35> LBL: h17 ARG0: x16 RSTR: h18 BODY: h19 ]
...             [ ord<36:42> LBL: h20 CARG: "2" ARG0: e22 [ e SF: prop TENSE:␣
↪untensed MOOD: indicative PROG: bool PERF: - ] ARG1: x16 ]
...             [ _factory_n_1<43:51> LBL: h20 ARG0: x16 ] >
...     HCONS: < h0 qeq h1 h6 qeq h4 h10 qeq h12 h18 qeq h20 >
...     ICONS: < > ]''')
```

Then the basic structure can be inspected as follows:

```
>>> m.top
'h0'
>>> len(m.predications)
10
```

These two attributes are the only two described by the `SemanticStructure` interface and subclasses then define additional data structures. For instance, `MRS` has several additional attributes:

```
>>> m.index
'e2'
>>> len(m.rels)  # m.rels is equivalent to m.predications
10
>>> len(m.hcons)
4
>>> len(m.icons)
0
>>> list(m.variables)
['e2', 'x3', 'h6', 'h7', 'x8', 'h10', 'h11', 'e13', 'e14', 'i15', 'x16', 'h18', 'h19',
↪ 'e22', 'h0', 'h1', 'h4', 'h12', 'h20', 'h5', 'h9', 'h17']
```

The basic interface for predications is defined by the `Predication` class:

```
>>> p = m.predications[2]  # for MRS, same as m.rels[2]
>>> p.id  # see note below
'e2'
>>> p.predicate
'_have_v_1'
>>> p.type
'e'
```

Note that while EDS and DMRS have unique ids for each node, MRS does not formally guarantee unique ids for each of its Elementary Predications, but PyDelphin creates one for each `EP` in an `MRS`. These ids are used for some methods on `SemanticStructure` instances, as exemplified in a later example.

For MRS, the *EP* subclass is used for predications, defining some additional attributes:

```
>>> p.label
'h1'
>>> p.iv  # intrinsic variable
'e2'
>>> p.args
{'ARG0': 'e2', 'ARG1': 'x3', 'ARG2': 'x8'}
```

SemanticStructure also defines methods for getting at information that may be implemented differently by subclasses. For instance, *MRS* and *EDS* define arguments (or edges) on their respective *Predication* objects, while *DMRS* lists them separately as *links*, but the *SemanticStructure.arguments* method works for all representations, and returns a dictionary mapping predication ids to lists of role-argument pairs for all *outgoing* arguments (*MRS* has `ARG0` intrinsic arguments and `CARG` constant arguments which are not represented as arguments in *EDS* and *DMRS*, so these are accessed separately).

```
>>> for id, args in m.arguments().items():
...     print(id, args)
...
x3 []
q3 [('RSTR', 'h6'), ('BODY', 'h7')]
e2 [('ARG1', 'x3'), ('ARG2', 'x8')]
q8 [('RSTR', 'h10'), ('BODY', 'h11')]
x8 []
e13 [('ARG1', 'e14'), ('ARG2', 'x8')]
e14 [('ARG1', 'i15'), ('ARG2', 'x16')]
q16 [('RSTR', 'h18'), ('BODY', 'h19')]
e22 [('ARG1', 'x16')]
x16 []
```

Testing for and listing quantifiers also happens at the semantic structure level as it is more reliable than testing individual predications:

```
>>> m.is_quantifier('x3')
False
>>> m.is_quantifier('q3')  # use id, not intrinsic variable
True
>>> for p, q in m.quantification_pairs():
...     if q is None:  # unquantified predication
...         print('{}:{} (none)'.format(p.id, p.predicate))
...     else:
...         print('{}:{} ({}:{})'.format(p.id, p.predicate, q.id, q.predicate))
...
x3:pron (q3:pronoun_q)
e2:_have_v_1 (none)
x8:_capital_n_1 (q8:_enough_q)
e13:with_p (none)
e14:_build_v_1 (none)
e22:ord (none)
x16:_factory_n_1 (q16:_a_q)
```

Morphosemantic properties can be retrieved by a predication's id:

```
>>> p = m.predications[2]
>>> m.properties(p.id)
{'SF': 'prop', 'TENSE': 'pres', 'MOOD': 'indicative', 'PROG': '-', 'PERF': '-'}
```

In *MRS*, they are also available via the *variables* attribute with the intrinsic variable of an EP:

```
>>> m.variables[p.iv]
{'SF': 'prop', 'TENSE': 'pres', 'MOOD': 'indicative', 'PROG': '-', 'PERF': '-'}
```

EDS and *DMRS* objects also implement the same attributes and methods (with their own relevant additions).

```
>>> from delphin import eds
>>> e = eds.from_mrs(m)
>>> len(e.predications) == len(e.nodes)
True
>>> e.nodes[2].predicate
'_have_v_1'
>>> for id, args in e.arguments().items():
...     print(id, args)
x3 []
_1 [('BV', 'x3')]
e2 [('ARG1', 'x3'), ('ARG2', 'x8')]
_2 [('BV', 'x8')]
x8 []
e13 [('ARG1', 'e14'), ('ARG2', 'x8')]
e14 [('ARG2', 'x16')]
_3 [('BV', 'x16')]
e22 [('ARG1', 'x16')]
x16 []
```

Note that there may be some differences in identifier forms or special role names (`BV` above for quantifiers).

3.2 Scoping Semantic Structures

MRS and DMRS are scoping semantic representations, meaning they encode the quantifier scope, although they do so rather differently. The `ScopingSemanticStructure` class normalizes an interface to the scoping information via some additional methods, such as for inspecting the labeled scopes:

```
>>> top, scopes = m.scopes()
>>> top  # the label of the top scope, not the top handle (MRS.top)
'h1'
>>> for label, predications in scopes.items():
...     print(label, [p.predicate for p in predications])
...
h4 ['pron']
h5 ['pronoun_q']
h1 ['_have_v_1']
h9 ['_enough_q']
h12 ['_capital_n_1', 'with_p', '_build_v_1']
h17 ['_a_q']
h20 ['ord', '_factory_n_1']
```

The scopal argument structure is also available:

```
>>> for id, args in m.scopal_arguments().items():
...     print(id, args)
...
x3 []
q3 [('RSTR', 'qeq', 'h4')]
e2 []
q8 [('RSTR', 'qeq', 'h12')]
```

(continues on next page)

(continued from previous page)

```
x8 []
e13 []
e14 []
q16 [('RSTR', 'qeq', 'h20')]
e22 []
x16 []
```

Note that unlike `arguments()`, these return triples whose second member is the scopal relationship between the id and the scope label.

DMRS works similarly:

```
>>> from delphin import dmrs
>>> d = dmrs.from_mrs(m)
>>> top, scopes = d.scopes()
>>> top
'h2'
>>> for label, predications in scopes.items():
...     print(label, [p.predicate for p in predications])
...
h0 ['pron']
h1 ['pronoun_q']
h2 ['_have_v_1']
h3 ['_enough_q']
h6 ['_build_v_1', '_capital_n_1', 'with_p']
h7 ['_a_q']
h9 ['_factory_n_1', 'ord']
```

Because DMRS does not natively have scope labels, they are generated by `DMRS.scopes`. It is thus recommended to pass these generated scopes to other methods rather than generating them over again, both for computational efficiency and consistency:

```
>>> for id, args in d.scopal_arguments(scopes=scopes).items():
...     print(id, args)
...
10000 []
10001 [('RSTR', 'qeq', 'h8')]
10002 []
10003 [('RSTR', 'qeq', 'h8')]
10004 []
10005 []
10006 []
10007 [('RSTR', 'qeq', 'h8')]
10008 []
10009 []
```

3.3 Well-formed Structures

While it is possible to manipulate and create `MRS`, `EDS`, and `DMRS` objects, there is no guarantee that these actions result in a well-formed semantic structure. Well-formedness is crucial for certain operations, such as realizing sentences with a grammar or converting between representations. The `delphin.mrs` module has a number of functions for testing various facets of well-formedness:

```
>>> mrs.is_connected(m)
True
>>> mrs.has_intrinsic_variable_property(m)
True
>>> mrs.plausibly_scopes(m)
True
>>> mrs.is_well_formed(m)
True
```

CHAPTER

FOUR

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.ace* module, which provides functions and classes for compiling grammars, parsing, transfer, and generation.

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

```
>>> from delphin import ace
```

4.1 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.

4.2 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 `interface` module.

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()))
...
5
```

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

4.3 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.
```

4.4 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.'
```

4.5 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 "co-processors" 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.interface.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.

4.6 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

FIVE

PYDELPHIN AT THE COMMAND LINE

PyDelphin is primarily a library for creating more complex software, but some functions are directly useful as commands. To facilitate this usage, the **delphin** command (**delphin.exe** on Windows) provides an entry point to a number of subcommands, including: *convert*, *select*, *mkprof*, *process*, *compare*, and *repp*. These subcommands are command-line front-ends to the functions defined in `delphin.commands`.

5.1 Usage

The **delphin** command becomes available when PyDelphin is *installed*.

```
$ delphin --help
usage: delphin [-h] [-V]  ...

PyDelphin command-line interface

optional arguments:
  -h, --help     show this help message and exit
  -V, --version  show program's version number and exit

available subcommands:

    convert      Convert DELPH-IN Semantics representations
    select       Select data from [incr tsdb()] test suites
    mkprof       Create [incr tsdb()] test suites
    process      Process [incr tsdb()] test suites using ACE
    compare      Compare MRS results across test suites
    repp         Tokenize sentences using REPP

$ delphin --version
delphin 1.0.0
```

PyDelphin developers may find it useful to run the command without installing, which is available via the `delphin.main` module:

```
~/pydelphin$ python3 -m delphin.main --version
delphin 1.0.0
```

This guide assumes you have installed PyDelphin and thus have the **delphin** command available.

5.2 Subcommands

5.2.1 convert

The **convert** subcommand enables conversion of various DELPH-IN Semantics representations. The `--from` and `--to` options select the source and target 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 and have delphin.highlight installed, 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 --indent
[ "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 *mrsprolog*:

```
$ echo '[ "It rains." TOP: h0 RELS: < [ _rain_v_1<3:8> LBL: h1 ARG0: e2 ] > HCONS: <␣
↪h0 qeq h1 > ]' \
> | delphin convert --to mrsprolog --indent
psoa(h0,
  [rel('_rain_v_1',h1,
       [attrval('ARG0',e2)])],
  hcons([qeq(h0,h1)]))
```

The full list of codecs that PyDelphin can use can be obtained with the `--list` option, which groups them by their representation and indicates if they can read (`r`) or write (`w`) the format.

```
$ delphin convert --list
DMRS
     dmrsjson        r/w
     dmrspenman      r/w
     dmrstikz        -/w
     dmrx            r/w
     simpledmrs      r/w
EDS
     eds             r/w
     edsjson         r/w
     edspenman       r/w
MRS
     ace             r/-
     indexedmrs      r/w
     mrsjson         r/w
     mrsprolog       -/w
     mrx             r/w
     simplemrs       r/w
```

Try `delphin convert --help` for more information.

5.2.2 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@
```

Try `delphin select --help` for more information.

5.2.3 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 \
>     --skeleton
>     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.

5.2.4 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
```

Try `delphin process -help` for more information.

See also:

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

5.2.5 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>
[..]
```

Try `delphin compare --help` for more information.

See also:

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

5.2.6 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 --format triple <<< "Abrams didn't chase␣
↪Browne."
Applied: !^(.+)$              \1
-Abrams didn't chase Browne.
+ Abrams didn't chase Browne.
Applied: !'           ’
- Abrams didn't chase Browne.
+ Abrams didn’t chase Browne.
Applied: !^(.+)$              \1
- Abrams didn’t chase Browne.
+  Abrams didn’t chase Browne.
Applied: !  +
-  Abrams didn’t chase Browne.
+ Abrams didn’t chase Browne.
Applied: !([^ ])(\.) ([])}”"’'... ]*)$               \1 \2 \3
- Abrams didn’t chase Browne.
+ Abrams didn’t chase Browne .
Applied: !([^ ])([nN])['’`]([tT])            \1 \2’\3
- Abrams didn’t chase Browne .
+ Abrams did n’t chase Browne .
Done: Abrams did n’t chase Browne .
(0, 6, Abrams)
(7, 10, did)
(10, 13, n’t)
(14, 19, chase)
(20, 26, Browne)
(26, 27, .)
```

When outputting to a TTY, the output will be colored in the “diff” format. The `--verbose` (or `-v`) option is also useful. With `-v`, warnings about invalid REPP patterns will be shown; with `-vv`, information about each REPP module called and the final pre-tokenization alignments are shown; and with `-vvv`, debug lines will be shown with every rule attempted.

Try `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

SIX

WORKING WITH [INCR TSDB()] TEST SUITES

[incr tsdb()] is the canonical software for managing **test suites**—collections of test items for judging the performance of an implemented grammar—within DELPH-IN. While the original purpose of test suites 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. Low-level operations on test suite databases are defined in `delphin.tsdb` while `delphin.itsdb` builds on top of `delphin.tsdb` to provide a more user-friendly API and support for processing (e.g, parsing) test suites.

There are several words in use to describe test suites:

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

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

test suite a general term for both skeletons and profiles

Also note that `delphin.itsdb` uses SQL-like terminology for database entities while `delphin.tsdb` usually uses the older relational database terms (to be consistent with [incr tsdb()]):

`tsdb` term	`itsdb` term	Casual term	Examples
Database	Test Suite	"profile"	`tsdb/gold/mrs/`, `tsdb/skeletons/mrs/`
"	Profile	"profile"	`tsdb/gold/mrs/`
"	Skeleton	"skeleton"	`tsdb/skeletons/mrs/`
Schema	Schema	"relations file"	`tsdb/gold/mrs/relations`
Field	Field	"field"	`i-input :string` (in a schema)
Relation	Table	"item file", etc.	`tsdb/gold/mrs/item`
Record	Row	"line"	`11@unknown@formal@none@1@S@It rained.@...`
Column	Column	"field", "column"	`i-id`

This guide covers the `delphin.itsdb` module and expects that you've imported it 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
- The Wikipedia entry on database terminology: https://en.wikipedia.org/wiki/Relational_database#Terminology

6.1 Loading and Inspecting Test Suites

Loading a test suite is as simple as creating a *TestSuite* object with the directory as its argument:

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

The *TestSuite* instance loads the database schema and uses it to inspect the data in tables. The schema can be inspected via the *TestSuite.schema* attribute:

```
>>> list(ts.schema)
['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.schema['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 test suite return the *Table* whose name is the given key. Note that the table name is as it appears in the schema and not necessarily 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.'
```

Iterating over a table yields rows from the table. A *Row* object stores the raw string data internally (accessed via *Row.data*), but upon iteration or column lookup it is cast depending on the datatype specified in the schema.

```
>>> row = next(iter(ts['item']))
>>> row.data
('11', 'unknown', 'formal', 'none', '1', 'S', 'It rained.', '', '', '', '1', '2',
↪'Det regnet.', 'oe', '15-10-2006')
>>> tuple(row)
(11, 'unknown', 'formal', 'none', 1, 'S', 'It rained.', None, None, None, 1, 2, 'Det␣
↪regnet.', 'oe', datetime.datetime(2006, 10, 15, 0, 0))
>>> row['i-input']
'It rained.'
```

The *Table.select* method allows for iterating over a restricted subset of columns:

```
>>> for row in ts['item'].select('i-id', 'i-input'):
...     print(tuple(row))
...
(11, 'It rained.')
(21, 'Abrams barked.')
(31, 'The window opened.')
[...]
```

6.2 Modifying Test Suite Data

Test suite data can be modified or extended by interacting with the *Table* instance. The *make_record()* function of *delphin.tsdb* may be useful for creating new items, or the *Table.update* method for modifying single rows.

```
>>> from delphin import tsdb
>>> items = ts['item']
>>> # find the next available i-id
>>> next_i_id = items[-1]['i-id'] + 1
>>> # define the data
>>> colmap = {'i-id': next_i_id, 'i-input': '...'}
>>> # add a new row
>>> items.append(tsdb.make_record(colmap, items.fields))
>>> # oops, forgot a field; reassign that last row
>>> colmap['i-wf'] = 0
>>> items[-1] = tsdb.make_record(colmap, items.fields))
>>> # oops it should be 1, just fix that one field
>>> items.update(-1, {'i-wf': 1})
>>> # write to disk
>>> ts.commit()
```

6.3 TSQL Queries Over Test Suites

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.tsql` module can help.

```
>>> from delphin import tsql
>>> for row in tsql.select('i-id mrs', ts):
...     print(tuple(row))
...
(11, '[ LTOP: h0 INDEX: e2 [ e SF: prop TENSE: past MOOD: indicative PROG: - PERF: -␣
↪] RELS: < [ _rain_v_1<3:10> LBL: h1 ARG0: e2 ] > HCONS: < h0 qeq h1 > ICONS: < > ]')
(21, '[ LTOP: 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 CARG: "Abrams" ARG0: x3 ]  [ _bark_v_1<7:14> LBL:␣
↪h1 ARG0: e2 ARG1: x3 ] > HCONS: < h0 qeq h1 h5 qeq h7 > ICONS: < > ]')
(31, '[ LTOP: h0 INDEX: e2 [ e SF: prop TENSE: past MOOD: indicative PROG: - PERF: -␣
↪] RELS: < [ _the_q<0:3> LBL: h4 ARG0: x3 [ x PERS: 3 NUM: sg IND: + ] RSTR: h5␣
↪BODY: h6 ]  [ _window_n_1<4:10> LBL: h7 ARG0: x3 ]  [ _open_v_1<11:18> LBL: h1␣
↪ARG0: e2 ARG1: x3 ] > HCONS: < h0 qeq h1 h5 qeq h7 > ICONS: < > ]')
[...]
```

See also:

- `delphin.tsql` module

6.4 Writing Test Suites to Disk

When modifying test suites as described above, the `TestSuite.commit` method is how the changes get written to disk. This is similar to how relational databases perform "transactions", but currently PyDelphin does not ensure consistency in the same way.

For more control over how data gets written to disk, see the `delphin.tsdb` module's `write()` and `write_database()` functions.

See also:

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

6.5 Processing Test Suites with ACE

PyDelphin has the ability to process test suites 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 test suite.

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

By default the processed data will be written to disk as it is processed so the in-memory *TestSuite* object doesn't get too large. The `buffer_size` parameter of *TestSuite.process* can be used to write to disk more or less frequently or not at all.

When doing generation or transfer the input to the processor is in the table that will be overwritten. To avoid loss of data, the `source` parameter takes another *TestSuite* instance that provides the inputs. The *delphin.commands.mkprof()* function is useful for creating an empty test suite for storing the results, but note that it expects the test suite paths instead of *TestSuite* instances.

```
>>> from delphin import commands
>>> src_path = '~/grammars/jacy/tsdb/current/mrs'
>>> tgt_path = '~/grammars/jacy/tsdb/current/mrs-gen'
>>> commands.mkprof(tgt_path, source=src_path)
    9067 bytes  relations
   15573 bytes  item
       0 bytes  analysis
[...]
>>> src_ts = itsdb.TestSuite(src_path)
>>> tgt_ts = itsdb.TestSuite(tgt_path)
>>> with ace.ACEGenerator('~/grammars/jacy/jacy-0.9.30.dat') as cpu:
...     tgt_ts.process(cpu, source=src_ts)
...
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]
[...]
```

6.6 Troubleshooting

`TSDBWarning: Invalid date field`

This warning occurs when PyDelphin tries to cast a value with the `:date` datatype when the raw value is not an acceptable date format (see *delphin.tsdb.cast()* for an explanation). Practically this means that the date will not be usable for things like TSQL conditions, but also note that it can cause data loss when writing a profile containing invalid dates to disk as PyDelphin will not write invalid data. Low-level operations that do not cast the value, such as from the *delphin.tsdb* module, may be able to write the raw string without data loss, but it is better to just fix the invalid dates.

CHAPTER

SEVEN

DEVELOPER GUIDE

This guide is for helping developers of modules in the `delphin` namespace or developers of PyDelphin itself.

7.1 PyDelphin Development Philosophy

PyDelphin aims to be a library that makes it easy for both newcomers to DELPH-IN and experienced researchers to make use of DELPH-IN resources. The following are the main priorities for PyDelphin development:

1. Implementations are correct and grammar-agnostic
2. Public APIs are documented
3. Public APIs are user-friendly
4. Code is tested
5. Code is legible
6. Code is computationally efficient

Note that grammar-agnosticism and correctness are the same point. Some DELPH-IN technologies were created with only one grammar or tool in mind, but PyDelphin will, as much as possible, implement structures and processes that are independent of any one tool or grammar. This means that PyDelphin implements according to specifications (e.g., research papers or wiki specifications), and creates those specifications if the technology is not sufficiently documented. For some concrete examples, the wikis for MRS, TDL, TSQL, and SEM-I (among others) were created, in part, to establish the specification for PyDelphin to implement. Much of the information in those wikis was pieced together from various places, such as other wikis, Lisp and C code, publications, and actual examples of the respective technologies. PyDelphin generally should *not* include novel and experimental techniques or representations (but it can certainly be *used* to create such things!).

The API documentation of PyDelphin is almost as important as the code itself. Every class, method, function, attribute, and module that is exposed to the user should be documented. The APIs should also follow conventions (such as those set by the Python Standard Library) to help the APIs stay natural and intuitive to the user. The API should try to be helpful to the user while being transparent about what it is doing.

The code of PyDelphin should be unit tested for a variety of expected (and some unintended) uses. The code should follow PEP-8 style guidelines and, going forward, make use of PEP-484 type annotations. The lowest priority (but a priority nonetheless) is for code to be computationally efficient. Software is more useful when it gives results quickly, but if users have a real need for efficient code they may want to look beyond Python.

7.2 Creating a New Plugin Module

The `delphin` package of PyDelphin is, as of version 1.0.0, a namespace package, which means that it is possible to create plugins under the `delphin` namespace.

7.2.1 Plugin Names

Plugin modules that define a single module or subpackage should be named `delphin.{name}` (e.g., delphin.highlight). If it includes more than one module or the plugin name doesn't strictly coincide with the project name, use `delphin-{{name}}` (e.g., delphin-latex).

7.2.2 Project Structure

The general project structure of a plugin module looks like this:

```
delphin.myplugin
├── delphin
│   └── myplugin.py
├── tests
│   └── test_myplugin.py
├── LICENSE
├── README.md
└── setup.py
```

The important thing to note is that the `delphin/` subdirectory does *not* contain an `__init__.py` file. If `myplugin.py` needed to be a package rather than a module, it could be a subdirectory of `delphin/` with an `__init__.py` file inside of it. Packages and modules under `delphin/` should not conflict with existing names in PyDelphin.

7.2.3 Plugin Versions

Each module should specify its version. The version should be included in `setup.py` as well as in the module as the `__version__` module constant.

7.3 Creating a New Codec Plugin

Creating serialization codec plugins is the same as for regular plugins, except that the module should go under `delphin/codecs/mycodec.py`, and neither `delphin/` nor `delphin/codecs/` should contain an `__init__.py`. The project could be dot-named `delphin.codecs.{{name}}` or something more generic with hyphen (such as the aforementioned delphin-latex).

In addition, the module should implement the *Codec API*, including the required module constant `CODEC_INFO`. If the module follows this API, it will be recognized by PyDelphin and appear in the list of available codecs when running `delphin convert --list` (see the *convert* command).

7.4 Defining a New Subcommand

Plugins can define subcommands that become available as **`delphin <subcommand>`** by creating a module in the `delphin.cli` namespace. Normally, the primary code of a plugin goes in the module of the `delphin` namespace and the `delphin.cli` module only defines a translation from command-line arguments to internal function calls.

See *delphin.cli* for more information about defining such modules.

7.5 Adding New Modules to PyDelphin

The modules that are included by default with the PyDelphin distribution should be generally useful and not include experimental features (see the *PyDelphin Development Philosophy*). With the understanding that in research software the line between "established" and "experimental" can get fuzzy, it might help to ask:

- *does this feature pertain to only one grammar?*
- *was this feature used for a one-off experiment?*

If the answer is *yes* to any of the above, then it might not be relevant for PyDelphin, but it is possible to create a plugin module, as described above, and distribute it on PyPI. One would only need to `pip install ...` to incorporate the new module into the `delphin` namespace.

If in fact users could benefit from including the module with PyDelphin proper, then one might petition the project maintainer to include the module in the next release of PyDelphin. In this case, please file an issue or pull request to request the merge.

7.6 Module Dependencies

Below is a listing of modules arranged into tiers by their dependencies. A "tier" is just a grouping here; there is no corresponding structure in the code except for the imports used in the modules. Each module within a tier only imports modules from tiers above it (imported modules, except for Tier 0 ones, are shown in parentheses after the module name).

It is good for a module to be conservative with its dependencies (i.e., descend to lower tiers). Module authors may consult this list to see on which tier their modules would fit.

If someone wants to take over maintainership of a PyDelphin module and spin it off as a separate repository, then modules without dependencies are the most eligible. For instance, if someone wants to take over responsibility for the *`delphin.mrs`* module, then they may want to also include the MRS codecs in their repository, or at least test the codecs to changes they make.

- Tier 0

- `delphin.__about__`
- *delphin.exceptions*
- `delphin.util`

- Tier 1
  - *delphin.derivation*
  - *delphin.hierarchy*
  - *delphin.interface* (soft dependencies on `tokens`, `derivation`, and `codecs`)
  - *delphin.lnk*
  - *delphin.predicate*
  - *delphin.variable*
- Tier 2
  - *delphin.ace* [`interface`]
  - *delphin.itsdb* [`interface`]
  - *delphin.sembase* [`lnk`]
  - *delphin.semi* [`hierarchy`, `predicate`]
  - *delphin.tfs* [`hierarchy`]
  - *delphin.tokens* [`lnk`]
  - *delphin.vpm* [`variable`]
  - *delphin.web.client* [`interface`]
- Tier 3
  - *delphin.repp* [`lnk`, `tokens`]
  - *delphin.scope* [`lnk`, `predicate`, `sembase`]
  - *delphin.tdl* [`tfs`]
  - *delphin.tsql* [`itsdb`]
- Tier 4
  - *delphin.dmrs* [`lnk`, `scope`, `sembase`, `variable`]
  - *delphin.eds* [`lnk`, `scope`, `sembase`, `variable`]
  - *delphin.mrs* [`lnk`, `predicate`, `scope`, `sembase`, `variable`]
- Tier 5
  - `delphin.codecs` [`dmrs`, `eds`, `mrs`, ...] (see *delphin.codecs*)
  - *delphin.web.server* [`ace`, `codecs`, `derivation`, `dmrs`, `eds`, `itsdb`, `tokens`]
- Tier 6
  - *delphin.commands* [`itsdb`, `lnk`, `semi`, `tsql`, ...]

CHAPTER

EIGHT

DELPHIN.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. The ACE software is required for the functionality in this module, but it is not included with PyDelphin. Pre-compiled binaries are available for Linux and MacOS at http://sweaglesw.org/linguistics/ace/, and 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, preferably, instantiate the class in a context manager so it is closed automatically when the relevant code has finished.

Interpreted responses are stored in a dictionary-like `Response` 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 `Result` 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`:

```
>>> from delphin import ace
>>> with ace.ACEParser('erg-2018-x86-64-0.9.30.dat') as parser:
...     response = parser.interact('A cat sleeps.')
...     print(response.result(0)['mrs'])
...     print(response.result(0).mrs())
...
[ LTOP: h0 INDEX: e2 [ e SF: prop TENSE: pres MOOD: indicative PROG: - PERF: - ]␣
↪RELS: < [ _a_q<0:1> LBL: h4 ARG0: x3 [ x PERS: 3 NUM: sg IND: + ] RSTR: h5 BODY: h6␣
↪]  [ _cat_n_1<2:5> LBL: h7 ARG0: x3 ]  [ _sleep_v_1<6:13> LBL: h1 ARG0: e2 ARG1: x3␣
↪] > HCONS: < h0 qeq h1 h5 qeq h7 > ICONS: < > ]
<MRS object (_a_q _cat_n_1 _sleep_v_1) at 140612036960072>
```

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.

8.1 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.ace.`**`compile`**(*cfg_path*, *out_path*, *executable=None*, *env=None*, *stdout=None*, *stderr=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
- **stdout** (*file, optional*) – stream used for ACE's stdout
- **stderr** (*file, optional*) – stream used for ACE's stderr

`delphin.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 *Response*

Example

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

`delphin.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 *Response*

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.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 *Response*

delphin.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 *Response*

delphin.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 *Response*

delphin.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 *Response*

8.2 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.ace.**ACEProcess**(*grm*, *cmdargs=None*, *executable=None*, *env=None*, *tsdbinfo=True*, *full_forest=False*, *stderr=None*)

Bases: *delphin.interface.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.

Note that not all arguments to this class are used by every subclass; the documentation for each subclass specifies which are available.

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
- **full_forest** (*bool*) – if `True` and *tsdbinfo* is `True`, output the full chart for each parse result
- **stderr** (*file*) – stream used for ACE's stderr

property 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 *Response*

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 *Response*

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.

property 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.

class delphin.ace.**ACEParser**(*grm*, *cmdargs=None*, *executable=None*, *env=None*, *tsdbinfo=True*, *full_forest=False*, *stderr=None*)
Bases: *delphin.ace.ACEProcess*

A class for managing parse requests with ACE.

See *ACEProcess* for initialization parameters.

class delphin.ace.**ACETransferer**(*grm*, *cmdargs=None*, *executable=None*, *env=None*, *stderr=None*)
Bases: *delphin.ace.ACEProcess*

A class for managing transfer requests with ACE.

See *ACEProcess* for initialization parameters.

class delphin.ace.**ACEGenerator**(*grm*, *cmdargs=None*, *executable=None*, *env=None*, *tsdbinfo=True*, *stderr=None*)
Bases: *delphin.ace.ACEProcess*

A class for managing realization requests with ACE.

See *ACEProcess* for initialization parameters.

8.3 Exceptions

exception delphin.ace.**ACEProcessError**(**args*, ***kwargs*)
Bases: *delphin.exceptions.PyDelphinException*

Raised when the ACE process has crashed and cannot be recovered.

8.4 ACE stdout Protocols

PyDelphin communicates with ACE via its "stdout protocols", which are the ways ACE's outputs are encoded across its stdout stream. There are several protocols that ACE uses and that this module supports:

- regular parsing
- parsing with ACE's `-tsdb-stdout` option

- parsing with `-tsdb-stdout` and `-itsdb-forest`
- transfer
- regular generation
- generation with ACE's `-tsdb-stdout` option

When a user interacts with ACE via the classes and functions in this module, responses will be interpreted and wrapped in `Response` objects, thus separating the user from the details of ACE's stdout protocols. Sometimes, however, the user will store or pipe ACE's output directly, such as when using the **delphin convert** *command* with **ace** at the command line. Even though ACE outputs MRSs using the common *SimpleMRS* format, additional content used in ACE's stdout protocols can complicate tasks such as format or represenation conversion. The user can provide some options to ACE (see http://moin.delph-in.net/AceOptions), such as **-T**, to filter the non-MRS content, but for convenience PyDelphin also provides the `ace` *codec*, available at `delphin.codecs.ace`. The codec ignores the non-MRS content in ACE's stdout so the user can use ACE output as a stream or as a corpus of MRS representations. For example:

```
.. code-block:: console
```

[~]$ ace -g erg.dat < sentences.txt | delphin convert –from ace

The codec does not support every stdout protocol that this module does. Those it does support are:

- regular parsing
- parsing with ACE's `-tsdb-stdout` option
- generation with ACE's `-tsdb-stdout` option

CHAPTER

NINE

DELPHIN.CLI

Command-line Interface Modules

The delphin.cli package is a namespace package for modules that define command-line interfaces. Each module under delphin.cli must have a dictionary named COMMAND_INFO and defined as follows:

```
COMMAND_INFO = {
    'name': 'command-name',             # Required
    'help': 'short help message',       # Optional
    'description': 'long description',  # Optional
    'parser': parser,                   # Required
}
```

The name field is the subcommand (e.g., **delphin command-name**) and the parser field is a argparse.ArgumentParser instance that specifies available arguments. Some common options, such as --verbose (-v), --quiet (-q), and --version (-V) will be created automatically by PyDelphin. This parser should also specify a func callable attribute that is called when the subcommand is used. Thus, the recommended way to create parser is as follows:

```
parser = argparse.ArgumentParser(add_help=False)
parser.set_defaults(func=my_function)
```

All of the default commands in *delphin.commands* define their command-line interface in the delphin.cli namespace.

CHAPTER

TEN

DELPHIN.CODECS

Serialization Codecs for Semantic Representations

The `delphin.codecs` package is a namespace package for modules used in the serialization and deserialization of semantic representations. All modules included in this namespace must follow the common API (based on Python's `pickle` and `json` modules) in order to work correctly with PyDelphin. This document describes that API.

10.1 Included Codecs

MRS:

10.1.1 delphin.codecs.simplemrs

Serialization functions for the SimpleMRS format.

SimpleMRS is a format for Minimal Recursion Semantics that aims to be readable equally by humans and machines.

Example:

- *The new chef whose soup accidentally spilled quit and left.*

```
[ TOP: h0
  INDEX: e2 [ e SF: prop TENSE: past MOOD: indicative PROG: - PERF: - ]
  RELS: < [ _the_q<0:3> LBL: h4 ARG0: x3 [ x PERS: 3 NUM: sg IND: + ] RSTR: h5 BODY: h6 ]
          [ _new_a_1<4:7> LBL: h7 ARG0: e8 [ e SF: prop TENSE: untensed MOOD: indicative PROG: bool PERF: - ] ARG1: x3 ]
          [ _chef_n_1<8:12> LBL: h7 ARG0: x3 ]
          [ def_explicit_q<13:18> LBL: h9 ARG0: x10 [ x PERS: 3 NUM: sg ] RSTR: h11 BODY: h12 ]
          [ poss<13:18> LBL: h13 ARG0: e14 [ e SF: prop TENSE: untensed MOOD: indicative PROG: - PERF: - ] ARG1: x10 ARG2: x3 ]
          [ _soup_n_1<19:23> LBL: h13 ARG0: x10 ]
          [ _accidental_a_1<24:36> LBL: h7 ARG0: e15 [ e SF: prop TENSE: untensed MOOD: indicative PROG: - PERF: - ] ARG1: e16 [ e SF: prop TENSE: past MOOD: indicative PROG: - PERF: - ] ]
          [ _spill_v_1<37:44> LBL: h7 ARG0: e16 ARG1: x10 ARG2: i17 ]
          [ _quit_v_1<45:49> LBL: h1 ARG0: e18 [ e SF: prop TENSE: past MOOD: indicative PROG: - PERF: - ] ARG1: x3 ARG2: i19 ]
          [ _and_c<50:53> LBL: h1 ARG0: e2 ARG1: e18 ARG2: e20 [ e SF: prop TENSE: past MOOD: indicative PROG: - PERF: - ] ]
          [ _leave_v_1<54:59> LBL: h1 ARG0: e20 ARG1: x3 ARG2: i21 ] >
  HCONS: < h0 qeq h1 h5 qeq h7 h11 qeq h13 > ]
```

Deserialization Functions

`delphin.codecs.simplemrs.load(source)`
See the `load()` codec API documentation.

`delphin.codecs.simplemrs.loads(s)`
See the `loads()` codec API documentation.

`delphin.codecs.simplemrs.decode(s)`
See the `decode()` codec API documentation.

Serialization Functions

`delphin.codecs.simplemrs.dump(ms, destination, properties=True, lnk=True, indent=False, encoding='utf-8')`
See the `dump()` codec API documentation.

`delphin.codecs.simplemrs.dumps(ms, properties=True, lnk=True, indent=False, encoding='utf-8')`
See the `dumps()` codec API documentation.

`delphin.codecs.simplemrs.encode(m, properties=True, lnk=True, indent=False, encoding='utf-8')`
See the `encode()` codec API documentation.

10.1.2 delphin.codecs.mrx

MRX (XML for MRS) serialization and deserialization.

Example:

- *The new chef whose soup accidentally spilled quit and left.*

```
<mrs cfrom="-1" cto="-1"><label vid="0" /><var sort="e" vid="2">
  <extrapair><path>SF</path><value>prop</value></extrapair>
  <extrapair><path>TENSE</path><value>past</value></extrapair>
  <extrapair><path>MOOD</path><value>indicative</value></extrapair>
  <extrapair><path>PROG</path><value>-</value></extrapair>
  <extrapair><path>PERF</path><value>-</value></extrapair></var>
  <ep cfrom="0" cto="3"><realpred lemma="the" pos="q" /><label vid="4" />
  <fvpair><rargname>ARG0</rargname><var sort="x" vid="3">
  <extrapair><path>PERS</path><value>3</value></extrapair>
  <extrapair><path>NUM</path><value>sg</value></extrapair>
  <extrapair><path>IND</path><value>+</value></extrapair></var></fvpair>
  <fvpair><rargname>RSTR</rargname><var sort="h" vid="5" /></fvpair>
  <fvpair><rargname>BODY</rargname><var sort="h" vid="6" /></fvpair></ep>
  <ep cfrom="4" cto="7"><realpred lemma="new" pos="a" sense="1" /><label vid="7" /
↪>
  <fvpair><rargname>ARG0</rargname><var sort="e" vid="8">
  <extrapair><path>SF</path><value>prop</value></extrapair>
  <extrapair><path>TENSE</path><value>untensed</value></extrapair>
  <extrapair><path>MOOD</path><value>indicative</value></extrapair>
  <extrapair><path>PROG</path><value>bool</value></extrapair>
  <extrapair><path>PERF</path><value>-</value></extrapair></var></fvpair>
  <fvpair><rargname>ARG1</rargname><var sort="x" vid="3" /></fvpair></ep>
  <ep cfrom="8" cto="12"><realpred lemma="chef" pos="n" sense="1" /><label vid="7
↪" />
  <fvpair><rargname>ARG0</rargname><var sort="x" vid="3" /></fvpair></ep>
```

(continues on next page)

(continued from previous page)

```
<ep cfrom="13" cto="18"><pred>def_explicit_q</pred><label vid="9" />
<fvpair><rargname>ARG0</rargname><var sort="x" vid="10">
<extrapair><path>PERS</path><value>3</value></extrapair>
<extrapair><path>NUM</path><value>sg</value></extrapair></var></fvpair>
<fvpair><rargname>RSTR</rargname><var sort="h" vid="11" /></fvpair>
<fvpair><rargname>BODY</rargname><var sort="h" vid="12" /></fvpair></ep>
<ep cfrom="13" cto="18"><pred>poss</pred><label vid="13" />
<fvpair><rargname>ARG0</rargname><var sort="e" vid="14">
<extrapair><path>SF</path><value>prop</value></extrapair>
<extrapair><path>TENSE</path><value>untensed</value></extrapair>
<extrapair><path>MOOD</path><value>indicative</value></extrapair>
<extrapair><path>PROG</path><value>-</value></extrapair>
<extrapair><path>PERF</path><value>-</value></extrapair></var></fvpair>
<fvpair><rargname>ARG1</rargname><var sort="x" vid="10" /></fvpair>
<fvpair><rargname>ARG2</rargname><var sort="x" vid="3" /></fvpair></ep>
<ep cfrom="19" cto="23"><realpred lemma="soup" pos="n" sense="1" /><label vid=
↪"13" />
<fvpair><rargname>ARG0</rargname><var sort="x" vid="10" /></fvpair></ep>
<ep cfrom="24" cto="36"><realpred lemma="accidental" pos="a" sense="1" /><label␣
↪vid="7" />
<fvpair><rargname>ARG0</rargname><var sort="e" vid="15">
<extrapair><path>SF</path><value>prop</value></extrapair>
<extrapair><path>TENSE</path><value>untensed</value></extrapair>
<extrapair><path>MOOD</path><value>indicative</value></extrapair>
<extrapair><path>PROG</path><value>-</value></extrapair>
<extrapair><path>PERF</path><value>-</value></extrapair></var></fvpair>
<fvpair><rargname>ARG1</rargname><var sort="e" vid="16">
<extrapair><path>SF</path><value>prop</value></extrapair>
<extrapair><path>TENSE</path><value>past</value></extrapair>
<extrapair><path>MOOD</path><value>indicative</value></extrapair>
<extrapair><path>PROG</path><value>-</value></extrapair>
<extrapair><path>PERF</path><value>-</value></extrapair></var></fvpair></ep>
<ep cfrom="37" cto="44"><realpred lemma="spill" pos="v" sense="1" /><label vid=
↪"7" />
<fvpair><rargname>ARG0</rargname><var sort="e" vid="16" /></fvpair>
<fvpair><rargname>ARG1</rargname><var sort="x" vid="10" /></fvpair>
<fvpair><rargname>ARG2</rargname><var sort="i" vid="17" /></fvpair></ep>
<ep cfrom="45" cto="49"><realpred lemma="quit" pos="v" sense="1" /><label vid="1
↪" />
<fvpair><rargname>ARG0</rargname><var sort="e" vid="18">
<extrapair><path>SF</path><value>prop</value></extrapair>
<extrapair><path>TENSE</path><value>past</value></extrapair>
<extrapair><path>MOOD</path><value>indicative</value></extrapair>
<extrapair><path>PROG</path><value>-</value></extrapair>
<extrapair><path>PERF</path><value>-</value></extrapair></var></fvpair>
<fvpair><rargname>ARG1</rargname><var sort="x" vid="3" /></fvpair>
<fvpair><rargname>ARG2</rargname><var sort="i" vid="19" /></fvpair></ep>
<ep cfrom="50" cto="53"><realpred lemma="and" pos="c" /><label vid="1" />
<fvpair><rargname>ARG0</rargname><var sort="e" vid="2" /></fvpair>
<fvpair><rargname>ARG1</rargname><var sort="e" vid="18" /></fvpair>
<fvpair><rargname>ARG2</rargname><var sort="e" vid="20">
<extrapair><path>SF</path><value>prop</value></extrapair>
<extrapair><path>TENSE</path><value>past</value></extrapair>
<extrapair><path>MOOD</path><value>indicative</value></extrapair>
<extrapair><path>PROG</path><value>-</value></extrapair>
<extrapair><path>PERF</path><value>-</value></extrapair></var></fvpair></ep>
<ep cfrom="54" cto="59"><realpred lemma="leave" pos="v" sense="1" /><label vid=
↪"1" />
```

(continues on next page)

(continued from previous page)

```
  <fvpair><rargname>ARG0</rargname><var sort="e" vid="20" /></fvpair>
  <fvpair><rargname>ARG1</rargname><var sort="x" vid="3" /></fvpair>
  <fvpair><rargname>ARG2</rargname><var sort="i" vid="21" /></fvpair></ep>
  <hcons hreln="qeq"><hi><var sort="h" vid="0" /></hi><lo><label vid="1" /></lo></
↪hcons>
  <hcons hreln="qeq"><hi><var sort="h" vid="5" /></hi><lo><label vid="7" /></lo></
↪hcons>
  <hcons hreln="qeq"><hi><var sort="h" vid="11" /></hi><lo><label vid="13" /></lo>
↪</hcons>
</mrs>
```

Module Constants

`delphin.codecs.mrx.`**`HEADER`**
    `'<mrs-list>'`

`delphin.codecs.mrx.`**`JOINER`**
    `''`

`delphin.codecs.mrx.`**`FOOTER`**
    `'</mrs-list>'`

Deserialization Functions

`delphin.codecs.mrx.`**`load`**`(`*source*`)`
    See the `load()` codec API documentation.

`delphin.codecs.mrx.`**`loads`**`(`*s*`)`
    See the `loads()` codec API documentation.

`delphin.codecs.mrx.`**`decode`**`(`*s*`)`
    See the `decode()` codec API documentation.

Serialization Functions

`delphin.codecs.mrx.`**`dump`**`(`*ms*, *destination*, *properties=True*, *lnk=True*, *indent=False*, *encoding='utf-8'*`)`
    See the `dump()` codec API documentation.

`delphin.codecs.mrx.`**`dumps`**`(`*ms*, *properties=True*, *lnk=True*, *indent=False*, *encoding='utf-8'*`)`
    See the `dumps()` codec API documentation.

`delphin.codecs.mrx.`**`encode`**`(`*m*, *properties=True*, *lnk=True*, *indent=False*, *encoding='utf-8'*`)`
    See the `encode()` codec API documentation.

10.1.3 delphin.codecs.indexedmrs

Serialization for the Indexed MRS format.

The Indexed MRS format does not include role names such as `ARG1`, `ARG2`, etc., so the order of the arguments in a predication is important. For this reason, serialization with the Indexed MRS format requires the use of a SEM-I (see the `delphin.semi` module).

Example:

- *The new chef whose soup accidentally spilled quit and left.*

```
< h0, e2:PROP:PAST:INDICATIVE:-:-,
  { h4:_the_q<0:3>(x3:3:SG:GENDER:+:PT, h5, h6),
    h7:_new_a_1<4:7>(e8:PROP:UNTENSED:INDICATIVE:BOOL:-, x3),
    h7:_chef_n_1<8:12>(x3),
    h9:def_explicit_q<13:18>(x10:3:SG:GENDER:BOOL:PT, h11, h12),
    h13:poss<13:18>(e14:PROP:UNTENSED:INDICATIVE:-:-, x10, x3),
    h13:_soup_n_1<19:23>(x10),
    h7:_accidental_a_1<24:36>(e15:PROP:UNTENSED:INDICATIVE:-:-,␣
↪e16:PROP:PAST:INDICATIVE:-:-),
    h7:_spill_v_1<37:44>(e16, x10, i17),
    h1:_quit_v_1<45:49>(e18:PROP:PAST:INDICATIVE:-:-, x3, i19),
    h1:_and_c<50:53>(e2, e18, e20:PROP:PAST:INDICATIVE:-:-),
    h1:_leave_v_1<54:59>(e20, x3, i21) },
  { h0 qeq h1,
    h5 qeq h7,
    h11 qeq h13 } >
```

Deserialization Functions

`delphin.codecs.indexedmrs.`**`load`**(*source*, *semi*)
: See the *`load()`* codec API documentation.

  Extensions:

  Parameters **`semi`** (`SemI`) – the semantic interface for the grammar that produced the MRS

`delphin.codecs.indexedmrs.`**`loads`**(*s*, *semi*)
: See the *`loads()`* codec API documentation.

  Extensions:

  Parameters **`semi`** (`SemI`) – the semantic interface for the grammar that produced the MRS

`delphin.codecs.indexedmrs.`**`decode`**(*s*, *semi*)
: See the *`decode()`* codec API documentation.

  Extensions:

  Parameters **`semi`** (`SemI`) – the semantic interface for the grammar that produced the MRS

Serialization Functions

`delphin.codecs.indexedmrs.`**`dump`**(*ms*, *destination*, *semi*, *properties=True*, *lnk=True*, *indent=False*, *encoding='utf-8'*)
: See the *`dump()`* codec API documentation.

  Extensions:

  Parameters **`semi`** (`SemI`) – the semantic interface for the grammar that produced the MRS

`delphin.codecs.indexedmrs.`**`dumps`**(*ms*, *semi*, *properties=True*, *lnk=True*, *indent=False*, *encoding='utf-8'*)
: See the *`dumps()`* codec API documentation.

  Extensions:

  Parameters **`semi`** (`SemI`) – the semantic interface for the grammar that produced the MRS `delphin.codecs.indexedmrs.`**`encode`**(*m*, *semi*, *properties=True*, *lnk=True*, *indent=False*, *encoding='utf-8'*)

See the `encode()` codec API documentation.

Extensions:

Parameters **`semi`** (`SemI`) – the semantic interface for the grammar that produced the MRS

10.1.4 delphin.codecs.mrsjson

MRS-JSON serialization and deserialization.

Example:

- *The new chef whose soup accidentally spilled quit and left.*

```
{
  "top": "h0",
  "index": "e2",
  "relations": [
    {
      "label": "h4",
      "predicate": "_the_q",
      "lnk": {"from": 0, "to": 3},
      "arguments": {"BODY": "h6", "RSTR": "h5", "ARG0": "x3"}
    },
    {
      "label": "h7",
      "predicate": "_new_a_1",
      "lnk": {"from": 4, "to": 7},
      "arguments": {"ARG1": "x3", "ARG0": "e8"}
    },
    {
      "label": "h7",
      "predicate": "_chef_n_1",
      "lnk": {"from": 8, "to": 12},
      "arguments": {"ARG0": "x3"}
    },
    {
      "label": "h9",
      "predicate": "def_explicit_q",
      "lnk": {"from": 13, "to": 18},
      "arguments": {"BODY": "h12", "RSTR": "h11", "ARG0": "x10"}
    },
    {
      "label": "h13",
      "predicate": "poss",
      "lnk": {"from": 13, "to": 18},
      "arguments": {"ARG1": "x10", "ARG2": "x3", "ARG0": "e14"}
    },
    {
      "label": "h13",
      "predicate": "_soup_n_1",
      "lnk": {"from": 19, "to": 23},
      "arguments": {"ARG0": "x10"}
    },
    {
      "label": "h7",
```

(continues on next page)

(continued from previous page)

```
      "predicate": "_accidental_a_1",
      "lnk": {"from": 24, "to": 36},
      "arguments": {"ARG1": "e16", "ARG0": "e15"}
    },
    {
      "label": "h7",
      "predicate": "_spill_v_1",
      "lnk": {"from": 37, "to": 44},
      "arguments": {"ARG1": "x10", "ARG2": "i17", "ARG0": "e16"}
    },
    {
      "label": "h1",
      "predicate": "_quit_v_1",
      "lnk": {"from": 45, "to": 49},
      "arguments": {"ARG1": "x3", "ARG2": "i19", "ARG0": "e18"}
    },
    {
      "label": "h1",
      "predicate": "_and_c",
      "lnk": {"from": 50, "to": 53},
      "arguments": {"ARG1": "e18", "ARG2": "e20", "ARG0": "e2"}
    },
    {
      "label": "h1",
      "predicate": "_leave_v_1",
      "lnk": {"from": 54, "to": 59},
      "arguments": {"ARG1": "x3", "ARG2": "i21", "ARG0": "e20"}
    }
  ],
  "constraints": [
    {"low": "h1", "high": "h0", "relation": "qeq"},
    {"low": "h7", "high": "h5", "relation": "qeq"},
    {"low": "h13", "high": "h11", "relation": "qeq"}
  ],
  "variables": {
    "h0": {"type": "h"},
    "h1": {"type": "h"},
    "e2": {"type": "e", "properties": {"MOOD": "indicative", "PROG": "-", "SF":
↪"prop", "PERF": "-", "TENSE": "past"}},
    "x3": {"type": "x", "properties": {"NUM": "sg", "PERS": "3", "IND": "+"}},
    "h4": {"type": "h"},
    "h6": {"type": "h"},
    "h5": {"type": "h"},
    "h7": {"type": "h"},
    "e8": {"type": "e", "properties": {"MOOD": "indicative", "PROG": "bool", "SF
↪": "prop", "PERF": "-", "TENSE": "untensed"}},
    "h9": {"type": "h"},
    "x10": {"type": "x", "properties": {"NUM": "sg", "PERS": "3"}},
    "h11": {"type": "h"},
    "h12": {"type": "h"},
    "h13": {"type": "h"},
    "e14": {"type": "e", "properties": {"MOOD": "indicative", "PROG": "-", "SF":
↪"prop", "PERF": "-", "TENSE": "untensed"}},
    "e15": {"type": "e", "properties": {"MOOD": "indicative", "PROG": "-", "SF":
↪"prop", "PERF": "-", "TENSE": "untensed"}},
    "e16": {"type": "e", "properties": {"MOOD": "indicative", "PROG": "-", "SF":
↪"prop", "PERF": "-", "TENSE": "past"}},
```

(continues on next page)

(continued from previous page)


```
    "i17": {"type": "i"},
    "e18": {"type": "e", "properties": {"MOOD": "indicative", "PROG": "-", "SF":
↪"prop", "PERF": "-", "TENSE": "past"}},
    "i19": {"type": "i"},
    "e20": {"type": "e", "properties": {"MOOD": "indicative", "PROG": "-", "SF":
↪"prop", "PERF": "-", "TENSE": "past"}},
    "i21": {"type": "i"}
  }
}
```


Module Constants

`delphin.codecs.mrsjson.`**`HEADER`**
    '['

`delphin.codecs.mrsjson.`**`JOINER`**
    ','

`delphin.codecs.mrsjson.`**`FOOTER`**
    ']'

Deserialization Functions

`delphin.codecs.mrsjson.`**`load`**(*source*)
    See the `load()` codec API documentation.

`delphin.codecs.mrsjson.`**`loads`**(*s*)
    See the `loads()` codec API documentation.

`delphin.codecs.mrsjson.`**`decode`**(*s*)
    See the `decode()` codec API documentation.

Serialization Functions

`delphin.codecs.mrsjson.`**`dump`**(*ms*, *destination*, *properties=True*, *lnk=True*, *indent=False*, *encoding='utf-8'*)
    See the `dump()` codec API documentation.

`delphin.codecs.mrsjson.`**`dumps`**(*ms*, *properties=True*, *lnk=True*, *indent=False*, *encoding='utf-8'*)
    See the `dumps()` codec API documentation.

`delphin.codecs.mrsjson.`**`encode`**(*m*, *properties=True*, *lnk=True*, *indent=False*, *encoding='utf-8'*)
    See the `encode()` codec API documentation.

Complementary Functions

`delphin.codecs.mrsjson.`**`from_dict`**(*d*)
    Decode a dictionary, as from `to_dict()`, into an MRS object.

`delphin.codecs.mrsjson.`**`to_dict`**(*mrs*, *properties=True*, *lnk=True*)
    Encode the MRS as a dictionary suitable for JSON serialization.

10.1.5 delphin.codecs.mrsprolog

Serialization functions for the MRS-Prolog format.

Example:

- *The new chef whose soup accidentally spilled quit and left.*

```
psoa(h0,e2,
  [rel('_the_q',h4,
       [attrval('ARG0',x3),
        attrval('RSTR',h5),
        attrval('BODY',h6)]),
   rel('_new_a_1',h7,
       [attrval('ARG0',e8),
        attrval('ARG1',x3)]),
   rel('_chef_n_1',h7,
       [attrval('ARG0',x3)]),
   rel('def_explicit_q',h9,
       [attrval('ARG0',x10),
        attrval('RSTR',h11),
        attrval('BODY',h12)]),
   rel('poss',h13,
       [attrval('ARG0',e14),
        attrval('ARG1',x10),
        attrval('ARG2',x3)]),
   rel('_soup_n_1',h13,
       [attrval('ARG0',x10)]),
   rel('_accidental_a_1',h7,
       [attrval('ARG0',e15),
        attrval('ARG1',e16)]),
   rel('_spill_v_1',h7,
       [attrval('ARG0',e16),
        attrval('ARG1',x10),
        attrval('ARG2',i17)]),
   rel('_quit_v_1',h1,
       [attrval('ARG0',e18),
        attrval('ARG1',x3),
        attrval('ARG2',i19)]),
   rel('_and_c',h1,
       [attrval('ARG0',e2),
        attrval('ARG1',e18),
        attrval('ARG2',e20)]),
   rel('_leave_v_1',h1,
       [attrval('ARG0',e20),
        attrval('ARG1',x3),
        attrval('ARG2',i21)])],
  hcons([qeq(h0,h1),qeq(h5,h7),qeq(h11,h13)]))
```

Serialization Functions

`delphin.codecs.mrsprolog.`**`dump`**(*ms*, *destination*, *properties=True*, *lnk=True*, *indent=False*, *encoding='utf-8'*)

See the `dump()` codec API documentation.

`delphin.codecs.mrsprolog.`**`dumps`**(*ms*, *properties=True*, *lnk=True*, *indent=False*, *encoding='utf-8'*)

See the `dumps()` codec API documentation.

delphin.codecs.mrsprolog.**encode**(*m*, *properties=True*, *lnk=True*, *indent=False*, *encoding='utf-8'*)

See the `encode()` codec API documentation.

10.1.6 delphin.codecs.ace

Deserialization of MRSs in ACE's stdout protocols.

Deserialization Functions

delphin.codecs.ace.**load**(*source*)

See the `load()` codec API documentation.

delphin.codecs.ace.**loads**(*s*)

See the `loads()` codec API documentation.

delphin.codecs.ace.**decode**(*s*)

See the `decode()` codec API documentation.

DMRS:

10.1.7 delphin.codecs.simpledmrs

Serialization for the SimpleDMRS format.

Example:

- *The new chef whose soup accidentally spilled quit and left.*

```
dmrs {
  ["The new chef whose soup accidentally spilled quit and left." top=10008␣
↪index=10009]
  10000 [_the_q<0:3>];
  10001 [_new_a_1<4:7> e SF=prop TENSE=untensed MOOD=indicative PROG=bool PERF=-];
  10002 [_chef_n_1<8:12> x PERS=3 NUM=sg IND=+];
  10003 [def_explicit_q<13:18>];
  10004 [poss<13:18> e SF=prop TENSE=untensed MOOD=indicative PROG=- PERF=-];
  10005 [_soup_n_1<19:23> x PERS=3 NUM=sg];
  10006 [_accidental_a_1<24:36> e SF=prop TENSE=untensed MOOD=indicative PROG=-␣
↪PERF=-];
  10007 [_spill_v_1<37:44> e SF=prop TENSE=past MOOD=indicative PROG=- PERF=-];
  10008 [_quit_v_1<45:49> e SF=prop TENSE=past MOOD=indicative PROG=- PERF=-];
  10009 [_and_c<50:53> e SF=prop TENSE=past MOOD=indicative PROG=- PERF=-];
  10010 [_leave_v_1<54:59> e SF=prop TENSE=past MOOD=indicative PROG=- PERF=-];
  10000:RSTR/H -> 10002;
  10001:ARG1/EQ -> 10002;
  10003:RSTR/H -> 10005;
  10004:ARG1/EQ -> 10005;
  10004:ARG2/NEQ -> 10002;
  10006:ARG1/EQ -> 10007;
  10007:ARG1/NEQ -> 10005;
  10008:ARG1/NEQ -> 10002;
  10009:ARG1/EQ -> 10008;
  10009:ARG2/EQ -> 10010;
  10010:ARG1/NEQ -> 10002;
  10007:MOD/EQ -> 10002;
```

(continues on next page)

(continued from previous page)

```
  10010:MOD/EQ -> 10008;
}
```

Deserialization Functions

delphin.codecs.simpledmrs.**load**(*source*)
See the `load()` codec API documentation.

delphin.codecs.simpledmrs.**loads**(*s*)
See the `loads()` codec API documentation.

delphin.codecs.simpledmrs.**decode**(*s*)
See the `decode()` codec API documentation.

Serialization Functions

delphin.codecs.simpledmrs.**dump**(*ms*, *destination*, *properties=True*, *lnk=True*, *indent=False*, *encoding='utf-8'*)
See the `dump()` codec API documentation.

delphin.codecs.simpledmrs.**dumps**(*ms*, *properties=True*, *lnk=True*, *indent=False*, *encoding='utf-8'*)
See the `dumps()` codec API documentation.

delphin.codecs.simpledmrs.**encode**(*m*, *properties=True*, *lnk=True*, *indent=False*, *encoding='utf-8'*)
See the `encode()` codec API documentation.

10.1.8 delphin.codecs.dmrx

DMRX (XML for DMRS) serialization and deserialization.

Example:

- *The new chef whose soup accidentally spilled quit and left.*

```
<dmrs cfrom="-1" cto="-1" index="10009" top="10008">
  <node cfrom="0" cto="3" nodeid="10000">
    <realpred lemma="the" pos="q" />
    <sortinfo />
  </node>
  <node cfrom="4" cto="7" nodeid="10001">
    <realpred lemma="new" pos="a" sense="1" />
    <sortinfo MOOD="indicative" PERF="-" PROG="bool" SF="prop" TENSE="untensed" 
↪cvarsort="e" />
  </node>
  <node cfrom="8" cto="12" nodeid="10002">
    <realpred lemma="chef" pos="n" sense="1" />
    <sortinfo IND="+" NUM="sg" PERS="3" cvarsort="x" />
  </node>
  <node cfrom="13" cto="18" nodeid="10003">
    <gpred>def_explicit_q</gpred>
    <sortinfo />
  </node>
  <node cfrom="13" cto="18" nodeid="10004">
```

(continues on next page)

(continued from previous page)

```
    <gpred>poss</gpred>
    <sortinfo MOOD="indicative" PERF="-" PROG="-" SF="prop" TENSE="untensed" cvarsort="e" />
  </node>
  <node cfrom="19" cto="23" nodeid="10005">
    <realpred lemma="soup" pos="n" sense="1" />
    <sortinfo NUM="sg" PERS="3" cvarsort="x" />
  </node>
  <node cfrom="24" cto="36" nodeid="10006">
    <realpred lemma="accidental" pos="a" sense="1" />
    <sortinfo MOOD="indicative" PERF="-" PROG="-" SF="prop" TENSE="untensed" cvarsort="e" />
  </node>
  <node cfrom="37" cto="44" nodeid="10007">
    <realpred lemma="spill" pos="v" sense="1" />
    <sortinfo MOOD="indicative" PERF="-" PROG="-" SF="prop" TENSE="past" cvarsort="e" />
  </node>
  <node cfrom="45" cto="49" nodeid="10008">
    <realpred lemma="quit" pos="v" sense="1" />
    <sortinfo MOOD="indicative" PERF="-" PROG="-" SF="prop" TENSE="past" cvarsort="e" />
  </node>
  <node cfrom="50" cto="53" nodeid="10009">
    <realpred lemma="and" pos="c" />
    <sortinfo MOOD="indicative" PERF="-" PROG="-" SF="prop" TENSE="past" cvarsort="e" />
  </node>
  <node cfrom="54" cto="59" nodeid="10010">
    <realpred lemma="leave" pos="v" sense="1" />
    <sortinfo MOOD="indicative" PERF="-" PROG="-" SF="prop" TENSE="past" cvarsort="e" />
  </node>
  <link from="10000" to="10002">
    <rargname>RSTR</rargname>
    <post>H</post>
  </link>
  <link from="10001" to="10002">
    <rargname>ARG1</rargname>
    <post>EQ</post>
  </link>
  <link from="10003" to="10005">
    <rargname>RSTR</rargname>
    <post>H</post>
  </link>
  <link from="10004" to="10005">
    <rargname>ARG1</rargname>
    <post>EQ</post>
  </link>
  <link from="10004" to="10002">
    <rargname>ARG2</rargname>
    <post>NEQ</post>
  </link>
  <link from="10006" to="10007">
    <rargname>ARG1</rargname>
    <post>EQ</post>
  </link>
```

(continues on next page)

(continued from previous page)

```
  <link from="10007" to="10005">
    <rargname>ARG1</rargname>
    <post>NEQ</post>
  </link>
  <link from="10008" to="10002">
    <rargname>ARG1</rargname>
    <post>NEQ</post>
  </link>
  <link from="10009" to="10008">
    <rargname>ARG1</rargname>
    <post>EQ</post>
  </link>
  <link from="10009" to="10010">
    <rargname>ARG2</rargname>
    <post>EQ</post>
  </link>
  <link from="10010" to="10002">
    <rargname>ARG1</rargname>
    <post>NEQ</post>
  </link>
  <link from="10007" to="10002">
    <rargname>MOD</rargname>
    <post>EQ</post>
  </link>
  <link from="10010" to="10008">
    <rargname>MOD</rargname>
    <post>EQ</post>
  </link>
</dmrs>
```

Module Constants

`delphin.codecs.dmrx.`**`HEADER`**
  '<dmrs-list>'

`delphin.codecs.dmrx.`**`JOINER`**
  ''

`delphin.codecs.dmrx.`**`FOOTER`**
  '</dmrs-list>'

Deserialization Functions

`delphin.codecs.dmrx.`**`load`**`(`*source*`)`
  See the `load()` codec API documentation.

`delphin.codecs.dmrx.`**`loads`**`(`*s*`)`
  See the `loads()` codec API documentation.

`delphin.codecs.dmrx.`**`decode`**`(`*s*`)`
  See the `decode()` codec API documentation.

Serialization Functions

delphin.codecs.dmrx.**dump**(*ms*, *destination*, *properties=True*, *lnk=True*, *indent=False*, *encoding='utf-8'*)
See the `dump()` codec API documentation.

delphin.codecs.dmrx.**dumps**(*ms*, *properties=True*, *lnk=True*, *indent=False*, *encoding='utf-8'*)
See the `dumps()` codec API documentation.

delphin.codecs.dmrx.**encode**(*m*, *properties=True*, *lnk=True*, *indent=False*, *encoding='utf-8'*)
See the `encode()` codec API documentation.

10.1.9 delphin.codecs.dmrsjson

DMRS-JSON serialization and deserialization.

Example:

- *The new chef whose soup accidentally spilled quit and left.*


```
{
  "top": 10008,
  "index": 10009,
  "nodes": [
    {
      "nodeid": 10000,
      "predicate": "_the_q",
      "lnk": {"from": 0, "to": 3}
    },
    {
      "nodeid": 10001,
      "predicate": "_new_a_1",
      "sortinfo": {"SF": "prop", "TENSE": "untensed", "MOOD": "indicative", "PROG
↪": "bool", "PERF": "-", "cvarsort": "e"},
      "lnk": {"from": 4, "to": 7}
    },
    {
      "nodeid": 10002,
      "predicate": "_chef_n_1",
      "sortinfo": {"PERS": "3", "NUM": "sg", "IND": "+", "cvarsort": "x"},
      "lnk": {"from": 8, "to": 12}
    },
    {
      "nodeid": 10003,
      "predicate": "def_explicit_q",
      "lnk": {"from": 13, "to": 18}
    },
    {
      "nodeid": 10004,
      "predicate": "poss",
      "sortinfo": {"SF": "prop", "TENSE": "untensed", "MOOD": "indicative", "PROG
↪": "-", "PERF": "-", "cvarsort": "e"},
      "lnk": {"from": 13, "to": 18}
    },
    {
      "nodeid": 10005,
      "predicate": "_soup_n_1",
      "sortinfo": {"PERS": "3", "NUM": "sg", "cvarsort": "x"},
```


(continues on next page)

(continued from previous page)

```
      "lnk": {"from": 19, "to": 23}
    },
    {
      "nodeid": 10006,
      "predicate": "_accidental_a_1",
      "sortinfo": {"SF": "prop", "TENSE": "untensed", "MOOD": "indicative", "PROG
↪": "-", "PERF": "-", "cvarsort": "e"},
      "lnk": {"from": 24, "to": 36}
    },
    {
      "nodeid": 10007,
      "predicate": "_spill_v_1",
      "sortinfo": {"SF": "prop", "TENSE": "past", "MOOD": "indicative", "PROG": "-
↪", "PERF": "-", "cvarsort": "e"},
      "lnk": {"from": 37, "to": 44}
    },
    {
      "nodeid": 10008,
      "predicate": "_quit_v_1",
      "sortinfo": {"SF": "prop", "TENSE": "past", "MOOD": "indicative", "PROG": "-
↪", "PERF": "-", "cvarsort": "e"},
      "lnk": {"from": 45, "to": 49}
    },
    {
      "nodeid": 10009,
      "predicate": "_and_c",
      "sortinfo": {"SF": "prop", "TENSE": "past", "MOOD": "indicative", "PROG": "-
↪", "PERF": "-", "cvarsort": "e"},
      "lnk": {"from": 50, "to": 53}
    },
    {
      "nodeid": 10010,
      "predicate": "_leave_v_1",
      "sortinfo": {"SF": "prop", "TENSE": "past", "MOOD": "indicative", "PROG": "-
↪", "PERF": "-", "cvarsort": "e"},
      "lnk": {"from": 54, "to": 59}
    }
  ],
  "links": [
    {"from": 10000, "to": 10002, "rargname": "RSTR", "post": "H"},
    {"from": 10001, "to": 10002, "rargname": "ARG1", "post": "EQ"},
    {"from": 10003, "to": 10005, "rargname": "RSTR", "post": "H"},
    {"from": 10004, "to": 10005, "rargname": "ARG1", "post": "EQ"},
    {"from": 10004, "to": 10002, "rargname": "ARG2", "post": "NEQ"},
    {"from": 10006, "to": 10007, "rargname": "ARG1", "post": "EQ"},
    {"from": 10007, "to": 10005, "rargname": "ARG1", "post": "NEQ"},
    {"from": 10008, "to": 10002, "rargname": "ARG1", "post": "NEQ"},
    {"from": 10009, "to": 10008, "rargname": "ARG1", "post": "EQ"},
    {"from": 10009, "to": 10010, "rargname": "ARG2", "post": "EQ"},
    {"from": 10010, "to": 10002, "rargname": "ARG1", "post": "NEQ"},
    {"from": 10007, "to": 10002, "rargname": "MOD", "post": "EQ"},
    {"from": 10010, "to": 10008, "rargname": "MOD", "post": "EQ"}
  ]
}
```

Module Constants

`delphin.codecs.dmrsjson.`**`HEADER`**
    '['

`delphin.codecs.dmrsjson.`**`JOINER`**
    ','

`delphin.codecs.dmrsjson.`**`FOOTER`**
    ']'

Deserialization Functions

`delphin.codecs.dmrsjson.`**`load`**(*source*)
    See the `load()` codec API documentation.

`delphin.codecs.dmrsjson.`**`loads`**(*s*)
    See the `loads()` codec API documentation.

`delphin.codecs.dmrsjson.`**`decode`**(*s*)
    See the `decode()` codec API documentation.

Serialization Functions

`delphin.codecs.dmrsjson.`**`dump`**(*ms*, *destination*, *properties=True*, *lnk=True*, *indent=False*, *encoding='utf-8'*)
    See the `dump()` codec API documentation.

`delphin.codecs.dmrsjson.`**`dumps`**(*ms*, *properties=True*, *lnk=True*, *indent=False*, *encoding='utf-8'*)
    See the `dumps()` codec API documentation.

`delphin.codecs.dmrsjson.`**`encode`**(*m*, *properties=True*, *lnk=True*, *indent=False*, *encoding='utf-8'*)
    See the `encode()` codec API documentation.

Complementary Functions

`delphin.codecs.dmrsjson.`**`from_dict`**(*d*)
    Decode a dictionary, as from `to_dict()`, into a DMRS object.

`delphin.codecs.dmrsjson.`**`to_dict`**(*d*, *properties=True*, *lnk=True*)
    Encode DMRS *d* as a dictionary suitable for JSON serialization.

10.1.10 delphin.codecs.dmrspenman

DMRS-PENMAN serialization and deserialization.

Example:

- *The new chef whose soup accidentally spilled quit and left.*

```
(e9 / _quit_v_1
  :lnk "<45:49>"
  :cvarsort e
  :sf prop
  :tense past
```

(continues on next page)

(continued from previous page)

```
  :mood indicative
  :prog -
  :perf -
  :ARG1-NEQ (x3 / _chef_n_1
    :lnk "<8:12>"
    :cvarsort x
    :pers 3
    :num sg
    :ind +
    :RSTR-H-of (q1 / _the_q
      :lnk "<0:3>")
    :ARG1-EQ-of (e2 / _new_a_1
      :lnk "<4:7>"
      :cvarsort e
      :sf prop
      :tense untensed
      :mood indicative
      :prog bool
      :perf -)
    :ARG2-NEQ-of (e5 / poss
      :lnk "<13:18>"
      :cvarsort e
      :sf prop
      :tense untensed
      :mood indicative
      :prog -
      :perf -
      :ARG1-EQ (x6 / _soup_n_1
        :lnk "<19:23>"
        :cvarsort x
        :pers 3
        :num sg
        :RSTR-H-of (q4 / def_explicit_q
          :lnk "<13:18>")))
    :MOD-EQ-of (e8 / _spill_v_1
      :lnk "<37:44>"
      :cvarsort e
      :sf prop
      :tense past
      :mood indicative
      :prog -
      :perf -
      :ARG1-NEQ x6
      :ARG1-EQ-of (e7 / _accidental_a_1
        :lnk "<24:36>"
        :cvarsort e
        :sf prop
        :tense untensed
        :mood indicative
        :prog -
        :perf -)))
  :ARG1-EQ-of (e10 / _and_c
    :lnk "<50:53>"
    :cvarsort e
    :sf prop
    :tense past
    :mood indicative
```

(continues on next page)

(continued from previous page)

```
    :prog -
    :perf -
    :ARG2-EQ (e11 / _leave_v_1
      :lnk "<54:59>"
      :cvarsort e
      :sf prop
      :tense past
      :mood indicative
      :prog -
      :perf -
      :ARG1-NEQ x3
      :MOD-EQ e9)))
```

Deserialization Functions

`delphin.codecs.dmrspenman.load(source)`
See the `load()` codec API documentation.

`delphin.codecs.dmrspenman.loads(s)`
See the `loads()` codec API documentation.

`delphin.codecs.dmrspenman.decode(s)`
See the `decode()` codec API documentation.

Serialization Functions

`delphin.codecs.dmrspenman.dump(ms, destination, properties=True, lnk=True, indent=False, encoding='utf-8')`
See the `dump()` codec API documentation.

`delphin.codecs.dmrspenman.dumps(ms, properties=True, lnk=True, indent=False, encoding='utf-8')`
See the `dumps()` codec API documentation.

`delphin.codecs.dmrspenman.encode(m, properties=True, lnk=True, indent=False, encoding='utf-8')`
See the `encode()` codec API documentation.

Complementary Functions

`delphin.codecs.dmrspenman.from_triples(triples)`
Decode triples, as from `to_triples()`, into a DMRS object.

`delphin.codecs.dmrspenman.to_triples(d, properties=True, lnk=True)`
Encode *d* as triples suitable for PENMAN serialization.

EDS:

10.1.11 delphin.codecs.eds

Serialization functions for the "native" EDS format.

Example:

- *The new chef whose soup accidentally spilled quit and left.*

```
{e18:
 _1:_the_q<0:3>[BV x3]
 e8:_new_a_1<4:7>{e SF prop, TENSE untensed, MOOD indicative, PROG bool, PERF -}
↪[ARG1 x3]
 x3:_chef_n_1<8:12>{x PERS 3, NUM sg, IND +}[]
 _2:def_explicit_q<13:18>[BV x10]
 e14:poss<13:18>{e SF prop, TENSE untensed, MOOD indicative, PROG -, PERF -}[ARG1␣
↪x10, ARG2 x3]
 x10:_soup_n_1<19:23>{x PERS 3, NUM sg}[]
 e15:_accidental_a_1<24:36>{e SF prop, TENSE untensed, MOOD indicative, PROG -,␣
↪PERF -}[ARG1 e16]
 e16:_spill_v_1<37:44>{e SF prop, TENSE past, MOOD indicative, PROG -, PERF -}
↪[ARG1 x10]
 e18:_quit_v_1<45:49>{e SF prop, TENSE past, MOOD indicative, PROG -, PERF -}
↪[ARG1 x3]
 e2:_and_c<50:53>{e SF prop, TENSE past, MOOD indicative, PROG -, PERF -}[ARG1␣
↪e18, ARG2 e20]
 e20:_leave_v_1<54:59>{e SF prop, TENSE past, MOOD indicative, PROG -, PERF -}
↪[ARG1 x3]
}
```

Deserialization Functions

`delphin.codecs.eds.`**`load`**`(`*source*`)`
: See the `load()` codec API documentation.

`delphin.codecs.eds.`**`loads`**`(`*s*`)`
: See the `loads()` codec API documentation.

`delphin.codecs.eds.`**`decode`**`(`*s*`)`
: See the `decode()` codec API documentation.

Serialization Functions

`delphin.codecs.eds.`**`dump`**`(`*ms, destination, properties=True, lnk=True, show_status=False, indent=False, encoding='utf-8'*`)`
: See the `dump()` codec API documentation.

Extensions:

Parameters **`show_status`** (`bool`) – if `True`, indicate disconnected components

`delphin.codecs.eds.`**`dumps`**`(`*ms, properties=True, lnk=True, show_status=False, indent=False, encoding='utf-8'*`)`
: See the `dumps()` codec API documentation.

Extensions:

Parameters **`show_status`** (`bool`) – if `True`, indicate disconnected components

`delphin.codecs.eds.`**`encode`**`(`*m, properties=True, lnk=True, show_status=False, indent=False, encoding='utf-8'*`)`
: See the `encode()` codec API documentation.

Extensions:

Parameters **`show_status`** (`bool`) – if `True`, indicate disconnected components

10.1.12 delphin.codecs.edsjson

EDS-JSON serialization and deserialization.

Example:

- *The new chef whose soup accidentally spilled quit and left.*

```
{
  "top": "e18",
  "nodes": {
    "_1": {
      "label": "_the_q",
      "edges": {"BV": "x3"},
      "lnk": {"from": 0, "to": 3}
    },
    "e8": {
      "label": "_new_a_1",
      "edges": {"ARG1": "x3"},
      "lnk": {"from": 4, "to": 7},
      "type": "e",
      "properties": {"SF": "prop", "TENSE": "untensed", "MOOD": "indicative",
↪"PROG": "bool", "PERF": "-"}
    },
    "x3": {
      "label": "_chef_n_1",
      "edges": {},
      "lnk": {"from": 8, "to": 12},
      "type": "x",
      "properties": {"PERS": "3", "NUM": "sg", "IND": "+"}
    },
    "_2": {
      "label": "def_explicit_q",
      "edges": {"BV": "x10"},
      "lnk": {"from": 13, "to": 18}
    },
    "e14": {
      "label": "poss",
      "edges": {"ARG1": "x10", "ARG2": "x3"},
      "lnk": {"from": 13, "to": 18},
      "type": "e",
      "properties": {"SF": "prop", "TENSE": "untensed", "MOOD": "indicative",
↪"PROG": "-", "PERF": "-"}
    },
    "x10": {
      "label": "_soup_n_1",
      "edges": {},
      "lnk": {"from": 19, "to": 23},
      "type": "x",
      "properties": {"PERS": "3", "NUM": "sg"}
    },
    "e15": {
      "label": "_accidental_a_1",
      "edges": {"ARG1": "e16"},
      "lnk": {"from": 24, "to": 36},
      "type": "e",
      "properties": {"SF": "prop", "TENSE": "untensed", "MOOD": "indicative",
↪"PROG": "-", "PERF": "-"}
    },
```

(continues on next page)

(continued from previous page)

```
    "e16": {
      "label": "_spill_v_1",
      "edges": {"ARG1": "x10"},
      "lnk": {"from": 37, "to": 44},
      "type": "e",
      "properties": {"SF": "prop", "TENSE": "past", "MOOD": "indicative", "PROG":
↪"-", "PERF": "-"}
    },
    "e18": {
      "label": "_quit_v_1",
      "edges": {"ARG1": "x3"},
      "lnk": {"from": 45, "to": 49},
      "type": "e",
      "properties": {"SF": "prop", "TENSE": "past", "MOOD": "indicative", "PROG":
↪"-", "PERF": "-"}
    },
    "e2": {
      "label": "_and_c",
      "edges": {"ARG1": "e18", "ARG2": "e20"},
      "lnk": {"from": 50, "to": 53},
      "type": "e",
      "properties": {"SF": "prop", "TENSE": "past", "MOOD": "indicative", "PROG":
↪"-", "PERF": "-"}
    },
    "e20": {
      "label": "_leave_v_1",
      "edges": {"ARG1": "x3"},
      "lnk": {"from": 54, "to": 59},
      "type": "e",
      "properties": {"SF": "prop", "TENSE": "past", "MOOD": "indicative", "PROG":
↪"-", "PERF": "-"}
    }
  }
}
```

Module Constants

`delphin.codecs.edsjson.`**`HEADER`**
    '['

`delphin.codecs.edsjson.`**`JOINER`**
    ','

`delphin.codecs.edsjson.`**`FOOTER`**
    ']'

Deserialization Functions

`delphin.codecs.edsjson.`**`load`**(*source*)
    See the `load()` codec API documentation.

`delphin.codecs.edsjson.`**`loads`**(*s*)
    See the `loads()` codec API documentation.

`delphin.codecs.edsjson.`**`decode`**(*s*)
    See the `decode()` codec API documentation.

Serialization Functions

`delphin.codecs.edsjson.dump(ms, destination, properties=True, lnk=True, indent=False, encoding='utf-8')`
: See the `dump()` codec API documentation.

`delphin.codecs.edsjson.dumps(ms, properties=True, lnk=True, indent=False, encoding='utf-8')`
: See the `dumps()` codec API documentation.

`delphin.codecs.edsjson.encode(m, properties=True, lnk=True, indent=False, encoding='utf-8')`
: See the `encode()` codec API documentation.

Complementary Functions

`delphin.codecs.edsjson.from_dict(d)`
: Decode a dictionary, as from `to_dict()`, into an EDS object.

`delphin.codecs.edsjson.to_dict(eds, properties=True, lnk=True)`
: Encode the EDS as a dictionary suitable for JSON serialization.

10.1.13 delphin.codecs.edspenman

EDS-PENMAN serialization and deserialization.

Example:

- *The new chef whose soup accidentally spilled quit and left.*

```
(e18 / _quit_v_1
  :lnk "<45:49>"
  :type e
  :sf prop
  :tense past
  :mood indicative
  :prog -
  :perf -
  :ARG1 (x3 / _chef_n_1
    :lnk "<8:12>"
    :type x
    :pers 3
    :num sg
    :ind +
    :BV-of (_1 / _the_q
      :lnk "<0:3>")
    :ARG1-of (e8 / _new_a_1
      :lnk "<4:7>"
      :type e
      :sf prop
      :tense untensed
      :mood indicative
      :prog bool
      :perf -)
    :ARG2-of (e14 / poss
      :lnk "<13:18>"
      :type e
      :sf prop
      :tense untensed
```

(continues on next page)

(continued from previous page)

```
      :mood indicative
      :prog -
      :perf -
      :ARG1 (x10 / _soup_n_1
        :lnk "<19:23>"
        :type x
        :pers 3
        :num sg
        :BV-of (_2 / def_explicit_q
          :lnk "<13:18>")
        :ARG1-of (e16 / _spill_v_1
          :lnk "<37:44>"
          :type e
          :sf prop
          :tense past
          :mood indicative
          :prog -
          :perf -
          :ARG1-of (e15 / _accidental_a_1
            :lnk "<24:36>"
            :type e
            :sf prop
            :tense untensed
            :mood indicative
            :prog -
            :perf -)))))
  :ARG1-of (e2 / _and_c
    :lnk "<50:53>"
    :type e
    :sf prop
    :tense past
    :mood indicative
    :prog -
    :perf -
    :ARG2 (e20 / _leave_v_1
      :lnk "<54:59>"
      :type e
      :sf prop
      :tense past
      :mood indicative
      :prog -
      :perf -
      :ARG1 x3)))
```

Deserialization Functions

delphin.codecs.edspenman.**load**(*source*)
: See the `load()` codec API documentation.

delphin.codecs.edspenman.**loads**(*s*)
: See the `loads()` codec API documentation.

delphin.codecs.edspenman.**decode**(*s*)
: See the `decode()` codec API documentation.

Serialization Functions

delphin.codecs.edspenman.**dump**(*ms*, *destination*, *properties=True*, *lnk=True*, *indent=False*, *encoding='utf-8'*)
See the *dump()* codec API documentation.

delphin.codecs.edspenman.**dumps**(*ms*, *properties=True*, *lnk=True*, *indent=False*, *encoding='utf-8'*)
See the *dumps()* codec API documentation.

delphin.codecs.edspenman.**encode**(*m*, *properties=True*, *lnk=True*, *indent=False*, *encoding='utf-8'*)
See the *encode()* codec API documentation.

Complementary Functions

delphin.codecs.edspenman.**from_triples**(*triples*)
Decode triples, as from *to_triples()*, into an EDS object.

delphin.codecs.edspenman.**to_triples**(*e*, *properties=True*, *lnk=True*)
Encode the Eds as triples suitable for PENMAN serialization.

10.2 Codec API

10.2.1 Module Constants

There is one required module constant for codecs: `CODEC_INFO`. Its purpose is primarily to specify which representation (MRS, DMRS, EDS) it serializes. A codec without `CODEC_INFO` will work for programmatic usage, but it will not work with the *delphin.commands.convert()* function or at the command line with the **delphin convert** command, which use the `representation` key in `CODEC_INFO` to determine when and how to convert representations.

CODEC_INFO
A dictionary containing information about the codec. While codec authors may put arbitrary data here, there are two keys used by PyDelphin's conversion features: `representation` and `description`. Only `representation` is required, and should be set to one of `mrs`, `dmrs`, or `eds`. For example, the `mrsjson` codec uses the following:

```
CODEC_INFO = {
    'representation': 'mrs',
    'description': 'JSON-serialized MRS for the Web API'
}
```

The following module constants are optional and are used to describe strings that must appear in valid documents when serializing multiple semantics representations at a time, as with *dump()* and *dumps()*. It is used by *delphin.commands.convert()* to provide a streaming serialization rather than dumping the entire file at once. If the values are not defined in the codec module, default values will be used.

HEADER
The string to output before any of semantic representations are serialized. For example, in *delphin.codecs.mrx*, the value of `HEADER` is `<mrs-list>`, and in the `delphin.codecs.dmrstikz` module of the delphin-latex plugin it is an entire LaTeX preamble followed by `\begin{document}`.

JOINER
The string used to join multiple serialized semantic representations. For example, in *delphin.codecs.mrsjson*, it is a comma (`,`) following JSON's syntax. Normally it is either an empty string, a space, or a newline, depending on the conventions for the format and if the `indent` argument is set.

FOOTER
The string to output after all semantic representations have been serialized. For example, in *delphin.codecs.mrx*, it is `</mrs-list>`, and in `delphin.codecs.dmrstikz` it is `\end{document}`.

10.2.2 Deserialization Functions

The deserialization functions *load()*, *loads()*, and *decode()* accept textual serializations and return the interpreted semantic representation. Both *load()* and *loads()* expect full documents (including headers and footers, such as `<mrs-list>` and `</mrs-list>` around a *mrx* serialization) and return lists of semantic structure objects. The *decode()* function expects single representations (without headers and footers) and returns a single semantic structure object.

Reading from a file or stream

load(*source*)
Deserialize and return semantic representations from *source*.

Parameters **source** – path-like object or file handle of a source containing serialized semantic representations

Return type list

Reading from a string

loads(*s*)
Deserialize and return semantic representations from string *s*.

Parameters **s** – string containing serialized semantic representations

Return type list

Decoding from a string

decode(*s*)
Deserialize and return the semantic representation from string *s*.

Parameters **s** – string containing a serialized semantic representation

Return type subclass of *delphin.sembase.SemanticStructure*

10.2.3 Serialization Functions

The serialization functions *dump()*, *dumps()*, and *encode()* take semantic representations as input as either return a string or print to a file or stream. Both *dump()* and *dumps()* will provide the appropriate *HEADER*, *JOINER*, and *FOOTER* values to make the result a valid document. The *encode()* function only serializes a single semantic representation, which is generally useful when working with single representations, but is also useful when headers and footers are not desired (e.g., if you want the *dmrx* representation of a DMRS without `<dmrs-list>` and `</dmrs-list>` surrounding it).

Writing to a file or stream

dump (*xs*, *destination*, *properties=True*, *lnk=True*, *indent=False*, *encoding='utf-8'*)
Serialize semantic representations in *xs* to *destination*.

Parameters

- **xs** – iterable of `SemanticStructure` objects to serialize
- **destination** – path-like object or file object where data will be written to
- **properties** (`bool`) – if `False`, suppress morphosemantic properties
- **lnk** (`bool`) – if `False`, suppress surface alignments and strings
- **indent** – if `True` or an integer value, add newlines and indentation; some codecs may support an integer value for `indent`, which specifies how many columns to indent
- **encoding** (`str`) – if *destination* is a filename, write to the file with the given encoding; otherwise it is ignored

Writing to a string

dumps (*xs*, *properties=True*, *lnk=True*, *indent=False*)
Serialize semantic representations in *xs* and return the string.

The arguments are interpreted as in `dump()`.

Return type str

Encoding to a string

encode (*x*, *properties=True*, *lnk=True*, *indent=False*)
Serialize single semantic representations *x* and return the string.

The arguments are interpreted as in `dump()`.

Return type str

10.2.4 Variations

All serialization codecs should use the function signatures above, but some variations are possible. Codecs should not remove any positional or keyword arguments from functions, but they can be ignored. If any new positional arguments are added, they should appear after the last positional argument in its function, before the keyword arguments. New keyword arguments may be added in any order. Finally, a codec may omit some functions entirely, such as for export-only codecs that do not provide `load()`, `loads()`, or `decode()`. The module constants `HEADER`, `JOINER`, and `FOOTER` are also optional. Here are some examples of variations in PyDelphin:

- `delphin.codecs.indexedmrs` requires a `semi` positional argument.
- `delphin.codecs.mrsjson`, `delphin.codecs.dmrsjson`, and `delphin.codecs.edsjson` introduce `to_dict()` and `from_dict()` functions in their public API as they may be generally useful.
- `delphin.codecs.dmrspenman` and `delphin.codecs.edspenman` introduce `to_triples()` and `from_triples()` functions in their public API.
- `delphin.codecs.eds` allows a `show_status` keyword argument to turn on graph connectedness markers on serialization.

- `delphin.codecs.mrsprolog` and `delphin.codecs.dmrstikz` are export-only codecs and do not provide `load()`, `loads()`, or `decode()` functions.
- `delphin.ace` is an import-only codec and does not provide `dump()`, `dumps()`, or `encode()` functions.

CHAPTER

ELEVEN

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.

11.1 convert

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

Convert between various DELPH-IN Semantics representations.

If *source_fmt* ends with `"-lines"`, then *path* must be an input file containing one representation per line to be read with the `decode()` function of the source codec. If *target_fmt* ends with `"-lines"`, then any `HEADER`, `JOINER`, or `FOOTER` defined by the target codec are ignored. The *source_fmt* and *target_fmt* arguments are then downcased and hyphens are removed to normalize the codec name.

Note: For syntax highlighting, delphin.highlight must be installed, and it is only available for select target formats.

Parameters

- **path** (`str, Path, open 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`)
- **lnk** (`bool`) – include lnk surface alignments and surface strings if `True` (default: `True`)
- **color** (`bool`) – apply syntax highlighting if `True` and *target_fmt* is `"simplemrs"` (default: `False`)
- **indent** (`int`) – specifies an explicit number of spaces for indentation
- **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`)
- **semi** – a `delphin.semi.SemI` object or path to a SEM-I (ignored if *target_fmt* is not `indexedmrs`)

Returns *str* – the converted representation

11.2 select

`delphin.commands.select(query, path, record_class=None)`

Select data from [incr tsdb()] test suites.

Parameters

- **query** (*str*) – TSQL select query (e.g., `'i-id i-input mrs'` or `'* from item where readings > 0'`)
- **path** (*str, Path*) – path to a TSDB test suite
- **record_class** – alternative class for records in the selection

Yields selected data from the test suite

11.3 mkprof

`delphin.commands.mkprof(destination, source=None, schema=None, where=None, delimiter=None, refresh=False, skeleton=False, full=False, gzip=False, quiet=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, refresh=True` – read data from *destination*
- `source=None, refresh=False` – read sentences from stdin
- `source="sents.txt"` – read sentences from `sents.txt`

The latter two require the *schema* parameter.

Parameters

- **destination** (*str, Path*) – path of the new testsuite
- **source** (*str, Path*) – path to a source testsuite or a file containing sentences; if not given and *refresh* is `False`, sentences are read from stdin
- **schema** (*str, Path*) – path to a relations file to use for the created testsuite; if `None` and *source* is a test suite, the schema of *source* is used
- **where** (*str*) – TSQL condition to filter records by; ignored if *source* is not a testsuite
- **delimiter** (*str*) – if given, split lines from *source* or stdin on the character *delimiter*; if *delimiter* is `"@"`, split using `delphin.tsdb.split()`; a header line with field names is required; ignored when the data source is not text lines
- **refresh** (*bool*) – if `True`, rewrite the data at *destination*; implies *full* is `True`; ignored if *source* is not `None`, best combined with *schema* or *gzip* (default: `False`)

- **skeleton** (*bool*) – if True, only write tsdb-core files (default: False)
- **full** (*bool*) – if True, copy all data from the source testsuite; ignored if the data source is not a testsuite or if *skeleton* is True (default: False)
- **gzip** (*bool*) – if True, non-empty tables will be compressed with gzip
- **quiet** (*bool*) – if True, don't print summary information

11.4 process

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

Process the [incr tsdb()] profile *testsuite* with *grammar*.

Inputs are read from *source* and results are written to *testsuite*. If *source* is None, it is set to *testsuite*. It is common for *source* to be None in parsing tasks, but it is not recommended for transfer or generation because the MRS field is both read from and written to for these tasks. If *source* points to a valid [incr tsdb()] profile and *testsuite* is a path to a non-existing location, the profile directory is created at that path.

The default task is parsing, but generation is done if *generate* is True and transfer is done if *transfer* is True; only one or neither may be True. Input data is extracted from *source* using the TSQL query *select*. If *select* is None, the default depends on the task:

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

Parameters

- **grammar** (*str, Path*) – path to a compiled grammar image
- **testsuite** (*str, Path*) – path to the destination [incr tsdb()] testsuite
- **source** (*str, Path*) – path to the source [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 select inputs with the specified result-id (transfer and generation)
- **gzip** (*bool*) – if True, non-empty tables will be compressed with gzip
- **stderr** (*file*) – stream for ACE's stderr

11.5 compare

delphin.commands.**compare**(*testsuite*, *gold*, *select='i-id i-input mrs'*)
Compare two [incr tsdb()] profiles.

Parameters

- **testsuite** (*str, Path, TestSuite*) – path to the test [incr tsdb()] testsuite or a `TestSuite` object
- **gold** (*str, Path, 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}
```

11.6 repp

delphin.commands.**repp**(*source*, *config=None*, *module=None*, *active=None*, *format=None*, *color=False*, *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

- **source** (*str, Path, open file*) – filename, open file, or stream of sentence inputs
- **config** (*str, Path*) – path to a PET REPP configuration (.set) file
- **module** (*str, Path*) – 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"`)
- **color** (*bool*) – apply syntax highlighting if `True` (default: `False`)
- **trace_level** (*int*) – if `0` no trace info is printed; if `1`, applied rules are printed, if greater than `1`, both applied and unapplied rules (in order) are printed (default: `0`)

11.7 Exceptions

exception delphin.commands.**CommandError**(**args*, ***kwargs*)
Raised on an invalid command call.

CHAPTER

TWELVE

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
      ("" 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.

12.1 Loading Derivation Data

There are two functions for loading derivations from either the UDF/UDX string representation or the dictionary representation: *from_string()* and *from_dict()*.

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

`delphin.derivation.`**`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

`delphin.derivation.`**`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

12.2 UDF/UDX Classes

There are four classes for representing derivation trees. The *Derivation* class is used to contain the entire tree, while *UDFNode*, *UDFTerminal*, and *UDFToken* represent individual nodes.

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.

A Derivation object is simply a *UDFNode* but as it is intended to represent an entire derivation tree it performs additional checks on instantiation if the top node is a root node, namely that the top node only has the *entity* attribute set, and that it has only one node on its *daughters* list.

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

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).

parent

The parent node in the tree, or `None` for the root. Note that this is not a regular UDF/UDX attribute but is added for convenience in traversing the tree.

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`.

internals()
Return the list of internal nodes.

Internal nodes are nodes above preterminals. In other words, the union of internals and preterminals is the set of nonterminal nodes.

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

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

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=('form', 'tokens', 'id', 'entity', 'score', 'start', 'end', 'daughters', 'head', 'type')*, *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

class delphin.derivation.**UDFTerminal**(*form*, *tokens=None*, *parent=None*)
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.

parent
The parent node in the tree. Note that this is not a regular UDF/UDX attribute but is added for convenience in traversing the tree.

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=('form', 'tokens', 'id', 'entity', 'score', 'start', 'end', 'daughters', 'head', 'type'), 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

class delphin.derivation.**UDFToken**(*id*, *tfs*)
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.

12.3 Exceptions

exception delphin.derivation.**DerivationSyntaxError**(*message=None*, *filename=None*, *lineno=None*, *offset=None*, *text=None*)

Bases: *delphin.exceptions.PyDelphinSyntaxError*

Raised when parsing an invalid UDF string.

CHAPTER

THIRTEEN

DELPHIN.DMRS

Dependency Minimal Recursion Semantics ([DMRS])

13.1 Serialization Formats

13.2 Module Constants

delphin.dmrs.**FIRST_NODE_ID**
The node identifier `10000` which is conventionally the first identifier used in a DMRS structure. This constant is mainly used for DMRS conversion or serialization.

delphin.dmrs.**RESTRICTION_ROLE**
The `RSTR` role used in links to select the restriction of a quantifier.

delphin.dmrs.**EQ_POST**
The `EQ` post-slash label on links that indicates the endpoints of a link share a scope.

delphin.dmrs.**NEQ_POST**
The `NEQ` post-slash label on links that indicates the endpoints of a link do not share a scope.

delphin.dmrs.**HEQ_POST**
The `HEQ` post-slash label on links that indicates the `start` node of a link immediately outscopes the `end` node.

delphin.dmrs.**H_POST**
The `H` post-slash label on links that indicates the `start` node of a link is qeq to the `end` node (i.e., `start` scopes over `end`, but not necessarily immediately).

delphin.dmrs.**CVARSORT**
The `cvarsort` dictionary key in `Node.sortinfo` that accesses the node's `type`.

13.3 Classes

class delphin.dmrs.**DMRS**(*top=None*, *index=None*, *nodes=None*, *links=None*, *lnk=None*, *surface=None*, *identifier=None*)
Bases: `delphin.scope.ScopingSemanticStructure`

Dependency Minimal Recursion Semantics (DMRS) class.

DMRS instances have a list of Node objects and a list of Link objects. The scopal top node may be set directly via a parameter or may be implicitly set via a `/H` Link from the special node id `0`. If both are given, the link is ignored. The non-scopal top (index) node may only be set via the *index* parameter.

Parameters

- **top** – the id of the scopal top node
- **index** – the id of the non-scopal top node
- **nodes** – an iterable of DMRS nodes
- **links** – an iterable of DMRS links
- **lnk** – surface alignment
- **surface** – surface string
- **identifier** – a discourse-utterance identifier

top
The scopal top node.

index
The non-scopal top node.

nodes
The list of Nodes (alias of *predications*).

links
The list of Links.

lnk
The surface alignment for the whole MRS.

surface
The surface string represented by the MRS.

identifier
A discourse-utterance identifier.

Example:

```
>>> rain = Node(10000, '_rain_v_1', type='e')
>>> heavy = Node(10001, '_heavy_a_1', type='e')
>>> arg1_link = Link(10000, 10001, role='ARG1', post='EQ')
>>> d = DMRS(top=10000, index=10000, [rain], [arg1_link])
```

arguments(*types=None*, *expressed=None*)
Return a mapping of the argument structure.

When *types* is used, any DMRS Links with `Link.attr` set to *H_POST* or *HEQ_POST* are considered to have a type of 'h', so one can exclude scopal arguments by omitting 'h' on *types*. Otherwise an argument's type is the *Node.type* of the link's target.

Parameters

- **types** – an iterable of predication types to include
- **expressed** – if `True`, only include arguments to expressed predications; if `False`, only include those unexpressed; if `None`, include both

Returns A mapping of predication ids to lists of (role, target) pairs for outgoing arguments for the predication.

is_quantifier(*id*)
Return `True` if *id* is the id of a quantifier node.

properties(*id*)
Return the morphosemantic properties for *id*.

quantification_pairs()

Return a list of (Quantifiee, Quantifier) pairs.

Both the Quantifier and Quantifiee are `Predication` objects, unless they do not quantify or are not quantified by anything, in which case they are `None`. In well-formed and complete structures, the quantifiee will never be `None`.

Example

```
>>> [(p.predicate, q.predicate)
...  for p, q in m.quantification_pairs()]
[('_dog_n_1', '_the_q'), ('_bark_v_1', None)]
```

scopal_arguments(*scopes=None*)

Return a mapping of the scopal argument structure.

The return value maps node ids to lists of scopal arguments as (role, scope_relation, target) triples. If *scopes* is given, the target is the scope label, otherwise it is the target node's id. Only links with a `Link.role` value are considered, so `MOD/EQ` links are not included as scopal arguments.

Parameters **scopes** – mapping of scope labels to lists of predications

Example

```
>>> d = DMRS(...)  # for "It doesn't rain.
>>> d.scopal_arguments()
{10000: [('ARG1', 'qeq', 10001)]}
>>> top, scopes = d.scopes()
>>> d.scopal_arguments(scopes=scopes)
{10000: [('ARG1', 'qeq', 'h2')]}
```

scopes()

Return a tuple containing the top label and the scope map.

Note that the top label is different from `top`, which the top node's id. If `top` does not select a top node, the `None` is returned for the top label.

The scope map is a dictionary mapping scope labels to the lists of predications sharing a scope.

class delphin.dmrs.**Node**(*id*, *predicate*, *type=None*, *properties=None*, *carg=None*, *lnk=None*, *surface=None*, *base=None*)

Bases: `delphin.sembase.Predication`

A DMRS node.

Nodes are very simple predications for DMRSs. Nodes don't have arguments or labels like `delphin.mrs.EP` objects, but they do have an attribute for CARGs and contain their vestigial variable type and properties in `sortinfo`.

Parameters

- **id** – node identifier
- **predicate** – semantic predicate
- **type** – node type (corresponds to the intrinsic variable type in MRS)
- **properties** – morphosemantic properties
- **carg** – constant value (e.g., for named entities)

- **lnk** – surface alignment
- **surface** – surface string
- **base** – base form

id
node identifier

predicate
semantic predicate

type
node type (corresponds to the intrinsic variable type in MRS)

properties
morphosemantic properties

sortinfo
properties with the node type at key "cvarsort"

carg
constant value (e.g., for named entities)

lnk
surface alignment

cfrom
surface alignment starting position

cto
surface alignment ending position

surface
surface string

base
base form

property sortinfo
Morphosemantic property mapping with cvarsort.

class delphin.dmrs.**Link**(*start*, *end*, *role*, *post*)
Bases: `object`

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

start
node id of the start of the Link

end
node id of the end of the Link

role
role of the argument

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

13.4 Module Functions

`delphin.dmrs.`**`from_mrs`**(*m*, *representative_priority=None*)

Create a DMRS by converting from MRS *m*.

In order for MRS to DMRS conversion to work, the MRS must satisfy the intrinsic variable property (see *`delphin.mrs.has_intrinsic_variable_property()`*).

Parameters

- **`m`** – the input MRS
- **`representative_priority`** – a function for ranking candidate representative nodes; see `scope.representatives()`

Returns DMRS

Raises **`DMRSError when conversion fails.`** –

13.5 Exceptions

`exception` `delphin.dmrs.`**`DMRSSyntaxError`**(*message=None*, *filename=None*, *lineno=None*, *offset=None*, *text=None*)

Bases: *`delphin.exceptions.PyDelphinSyntaxError`*

Raised when an invalid DMRS serialization is encountered.

CHAPTER

FOURTEEN

DELPHIN.EDS

Elementary Dependency Structures ([EDS])

14.1 Serialization Formats

14.2 Module Constants

delphin.eds.**BOUND_VARIABLE_ROLE**
: The `BV` role used in edges to select the identifier of the node restricted by the quantifier.

delphin.eds.**PREDICATE_MODIFIER_ROLE**
: The `ARG1` role used as a default role when inserting edges for predicate modification.

14.3 Classes

class delphin.eds.**EDS**(*top=None*, *nodes=None*, *lnk=None*, *surface=None*, *identifier=None*)

Bases: *delphin.sembase.SemanticStructure*

An Elementary Dependency Structure (EDS) instance.

EDS are semantic structures deriving from MRS, but they are not interconvertible with MRS as the do not encode a notion of quantifier scope.

Parameters

- **top** – the id of the graph's top node
- **nodes** – an iterable of EDS nodes
- **lnk** – surface alignment
- **surface** – surface string
- **identifier** – a discourse-utterance identifier

arguments(*types=None*)

Return a mapping of the argument structure.

Parameters

- **types** – an iterable of predication types to include
- **expressed** – if `True`, only include arguments to expressed predications; if `False`, only include those unexpressed; if `None`, include both

Returns A mapping of predication ids to lists of (role, target) pairs for outgoing arguments for the predication.

property edges
The list of all edges.

is_quantifier(*id*)
Return `True` if *id* is the id of a quantifier node.

property nodes
Alias of `predications`.

properties(*id*)
Return the morphosemantic properties for *id*.

quantification_pairs()
Return a list of (Quantifiee, Quantifier) pairs.

Both the Quantifier and Quantifiee are `Predication` objects, unless they do not quantify or are not quantified by anything, in which case they are `None`. In well-formed and complete structures, the quantifiee will never be `None`.

Example

```
>>> [(p.predicate, q.predicate)
...  for p, q in m.quantification_pairs()]
[('_dog_n_1', '_the_q'), ('_bark_v_1', None)]
```

class delphin.eds.**Node**(*id*, *predicate*, *type=None*, *edges=None*, *properties=None*, *carg=None*, *lnk=None*, *surface=None*, *base=None*)
Bases: *delphin.sembase.Predication*

An EDS node.

Parameters

- **id** – node identifier
- **predicate** – semantic predicate
- **type** – node type (corresponds to the intrinsic variable type in MRS)
- **edges** – mapping of outgoing edge roles to target identifiers
- **properties** – morphosemantic properties
- **carg** – constant value (e.g., for named entities)
- **lnk** – surface alignment
- **surface** – surface string
- **base** – base form

id
node identifier

predicate
semantic predicate

type
node type (corresponds to the intrinsic variable type in MRS)

edges
mapping of outgoing edge roles to target identifiers

properties
morphosemantic properties

carg
constant value (e.g., for named entities)

lnk
surface alignment

cfrom
surface alignment starting position

cto
surface alignment ending position

surface
surface string

base
base form

14.4 Module Functions

delphin.eds.**from_mrs**(*m*, *predicate_modifiers=False*, *unique_ids=True*, *representative_priority=None*)
Create an EDS by converting from MRS *m*.

In order for MRS to EDS conversion to work, the MRS must satisfy the intrinsic variable property (see *delphin.mrs.has_intrinsic_variable_property()*).

Parameters

- **m** – the input MRS
- **predicate_modifiers** – if `True`, include predicate-modifier edges; if `False`, only include basic dependencies; if a callable, then call on the converted EDS before creating unique ids (if `unique_ids=True`)
- **unique_ids** – if `True`, recompute node identifiers to be unique by the LKB's method; note that ids from *m* should already be unique by PyDelphin's method
- **representative_priority** – a function for ranking candidate representative nodes; see `scope.representatives()`

Returns EDS

Raises ***EDSError*** – when conversion fails.

delphin.eds.**find_predicate_modifiers**(*e*, *m*, *representatives=None*)
Return an argument structure mapping for predicate-modifier edges.

In EDS, predicate modifiers are edges that describe a relation between predications in the original MRS that is not evident on the regular and scopal arguments. In practice these are EPs that share a scope but do not select any other EPs within their scope, such as when quantifiers are modified ("nearly every...") or with relative clauses ("the chef whose soup spilled..."). These are almost the same as the MOD/EQ links of DMRS, except that predicate modifiers have more restrictions on their usage, mainly due to their using a standard role (`ARG1`) instead of an idiosyncratic one.

Generally users won't call this function directly, but by calling `from_mrs()` with `predicate_modifiers=True`, but it is visible here in case users want to inspect its results separately from MRS-to-EDS conversion. Note that when calling it separately, *e* should use the same predication ids as *m* (by calling `from_mrs()` with `unique_ids=False`). Also, users may define their own function with the same signature and return type and use it in place of this one. See `from_mrs()` for details.

Parameters

- **e** – the EDS converted from *m* as by calling `from_mrs()` with `predicate_modifiers=False` and `unique_ids=False`, used to determine if parts of the graph are connected
- **m** – the source MRS
- **representatives** – the scope representatives; this argument is mainly to prevent `delphin.scope.representatives()` from being called twice on *m*

Returns A dictionary mapping source node identifiers to role-to-argument dictionaries of any additional predicate-modifier edges.

Examples

```
>>> e = eds.from_mrs(m, predicate_modifiers=False)
>>> print(eds.find_predicate_modifiers(e.argument_structure(), m)
{'e5': {'ARG1': '_1'}}
```

`delphin.eds.make_ids_unique(e, m)`

Recompute the node identifiers in EDS *e* to be unique.

MRS objects used in conversion to EDS already have unique predication ids, but they are created according to PyDelphin's method rather than the LKB's method, namely with regard to quantifiers and MRSs that do not have the intrinsic variable property. This function recomputes unique EDS node identifiers by the LKB's method.

Note: This function works in-place on *e* and returns nothing.

Parameters

- **e** – an EDS converted from MRS *m*, as from `from_mrs()` with `unique_ids=False`
- **m** – the MRS from which *e* was converted

14.5 Exceptions

exception `delphin.eds.EDSError(*args, **kwargs)`

Bases: `delphin.exceptions.PyDelphinException`

Raises on invalid EDS operations.

exception `delphin.eds.EDSSyntaxError(message=None, filename=None, lineno=None, offset=None, text=None)`

Bases: `delphin.exceptions.PyDelphinSyntaxError`

Raised when an invalid EDS string is encountered.

CHAPTER

FIFTEEN

DELPHIN.EXCEPTIONS

Basic exception and warning classes for PyDelphin.

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

exception delphin.exceptions.**PyDelphinSyntaxError**(*message=None*, *filename=None*, *lineno=None*, *offset=None*, *text=None*)

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

CHAPTER

SIXTEEN

DELPHIN.HIERARCHY

Basic support for hierarchies.

This module defines the *MultiHierarchy* class for multiply-inheriting hierarchies. This class manages the insertion of new nodes into the hierarchy via the class constructor or the *MultiHierarchy.update()* method, normalizing node identifiers (if a suitable normalization function is provided at instantiation), and inserting nodes in the appropriate order. It checks for some kinds of ill-formed hierarchies, such as cycles and redundant parentage and provides methods for testing for node compatibility and subsumption. For convenience, arbitrary data may be associated with node identifiers.

While the class may be used directly, it is mainly used to support the *TypeHierarchy* class and the predicate, property, and variable hierarchies of *SemI* instances.

16.1 Classes

class delphin.hierarchy.**MultiHierarchy**(*top*, *hierarchy=None*, *data=None*, *normalize_identifier=None*)

A Multiply-inheriting Hierarchy.

Hierarchies may be constructed when instantiating the class or via the *update()* method using a dictionary mapping identifiers to parents. In both cases, the parents may be a string of whitespace-separated parent identifiers or a tuple of (possibly non-string) identifiers. Also, both methods may take a `data` parameter which accepts a mapping from identifiers to arbitrary data. Data for identifiers may be get and set individually with dictionary key-access.

```
>>> h = MultiHierarchy('*top*', {'food': '*top*',
...                              'utensil': '*top*'})
>>> th.update({'fruit': 'food', 'apple': 'fruit'})
>>> th['apple'] = 'info about apples'
>>> th.update({'knife': 'utensil'},
...           data={'knife': 'info about knives'})
>>> th.update({'vegetable': 'food', 'tomato': 'fruit vegetable'})
```

In some ways the MultiHierarchy behaves like a dictionary, but it is not a subclass of `dict` and does not implement all its methods. Also note that some methods ignore the top node, which make certain actions easier:

```
>>> h = Hierarchy('*top*', {'a': '*top*', 'b': 'a', 'c': 'a'})
>>> len(h)
3
>>> list(h)
['a', 'b', 'c']
>>> Hierarchy('*top*', {id: h.parents(id) for id in h}) == h
True
```

But others do not ignore the top node, namely those where you can request it specifically:

```
>>> '*top*' in h
True
>>> print(h['*top*'])
None
>>> h.children('*top*')
{'a'}
```

Parameters

- **top** – the unique top identifier
- **hierarchy** – a mapping of node identifiers to parents (see description above concerning the possible parent values)
- **data** – a mapping of node identifiers to arbitrary data
- **normalize_identifier** – a unary function used to normalize identifiers (e.g., case normalization)

top
the hierarchy's top node identifier

ancestors (*identifier*)
Return the ancestors of *identifier*.

children (*identifier*)
Return the immediate children of *identifier*.

compatible (*a*, *b*)
Return `True` if node *a* is compatible with node *b*.

In a multiply-inheriting hierarchy, node compatibility means that two nodes share a common descendant. It is a commutative operation, so `compatible(a, b) == compatible(b, a)`. Note that in a singly-inheriting hierarchy, two nodes are never compatible by this metric.

Parameters

- **a** – a node identifier
- **b** – a node identifier

Examples

```
>>> h = MultiHierarchy('*top*', {'a': '*top*',
...                              'b': '*top*'})
>>> h.compatible('a', 'b')
False
>>> h.update({'c': 'a b'})
>>> h.compatible('a', 'b')
True
```

descendants (*identifier*)
Return the descendants of *identifier*.

items ()
Return the (identifier, data) pairs excluding the top node.

parents (*identifier*)
Return the immediate parents of *identifier*.

subsumes (*a*, *b*)
Return `True` if node *a* subsumes node *b*.

A node is subsumed by the other if it is a descendant of the other node or if it is the other node. It is not a commutative operation, so `subsumes(a, b) != subsumes(b, a)`, except for the case where `a == b`.

Parameters

- **a** – a node identifier
- **b** – a node identifier

Examples

```
>>> h = MultiHierarchy('*top*', {'a': '*top*',
...                              'b': '*top*',
...                              'c': 'b'})
>>> all(h.subsumes(h.top, x) for x in h)
True
>>> h.subsumes('a', h.top)
False
>>> h.subsumes('a', 'b')
False
>>> h.subsumes('b', 'c')
True
```

update (*subhierarchy=None*, *data=None*)
Incorporate *subhierarchy* and *data* into the hierarchy.

This method ensures that nodes are inserted in an order that does not result in an intermediate state being disconnected or cyclic, and raises an error if it cannot avoid such a state due to *subhierarchy* being invalid when inserted into the main hierarchy. Updates are atomic, so *subhierarchy* and *data* will not be partially applied if there is an error in the middle of the operation.

Parameters

- **subhierarchy** – mapping of node identifiers to parents
- **data** – mapping of node identifiers to data objects

Raises *HierarchyError* – when *subhierarchy* or *data* cannot be incorporated into the hierarchy

Examples

```
>>> h = MultiHierarchy('*top*')
>>> h.update({'a': '*top*'})
>>> h.update({'b': '*top*'}, data={'b': 5})
>>> h.update(data={'a': 3})
>>> h['b'] - h['a']
2
```

validate_update (*subhierarchy*, *data*)
Check if the update can apply to the current hierarchy.

This method returns (*subhierarchy*, *data*) with normalized identifiers if the update is valid, otherwise it will raise a `HierarchyError`.

Raises **`HierarchyError`** – when the update is invalid

16.2 Exceptions

exception `delphin.hierarchy.`**`HierarchyError`**(**args*, ***kwargs*)

Bases: `delphin.exceptions.PyDelphinException`

Raised for invalid operations on hierarchies.

CHAPTER

SEVENTEEN

DELPHIN.INTERFACE

Interfaces for external data providers.

This module manages the communication between data providers, namely processors like ACE or remote services like the DELPH-IN Web API, and user code or storage backends, namely [incr tsdb()] *test suites*. An interface sends requests to a provider, then receives and interprets the response. The interface may also detect and deserialize supported DELPH-IN formats if the appropriate modules are available.

17.1 Classes

class delphin.interface.**Processor**

Base class for processors.

This class defines the basic interface for all PyDelphin processors, such as `ACEProcess` and `Client`. 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

class delphin.interface.**Response**

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

result(*i*)

Return a Result object for the result *i*.

results()

Return Result objects for each result.

tokens(*tokenset='internal'*)

Interpret and return a YYTokenLattice object.

If *tokenset* is a key under the `tokens` key of the response, interpret its value as a `YYTokenLattice` from a valid YY serialization or from a dictionary. If *tokenset* is not available, return `None`.

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

Returns `YYTokenLattice`

Raises ***InterfaceError*** – when the value is an unsupported type or `delphin.tokens` is unavailble

class `delphin.interface.`**`Result`**

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

derivation()

Interpret and return a Derivation object.

If `delphin.derivation` is available and the value of the `derivation` key in the result dictionary is a valid UDF string or a dictionary, return the interpeted Derivation object. If there is no 'derivation' key in the result, return `None`.

Raises ***InterfaceError*** – when the value is an unsupported type or `delphin.derivation` is unavailable

dmrs()

Interpret and return a Dmrs object.

If `delphin.codecs.dmrsjson` is available and the value of the `dmrs` key in the result is a dictionary, return the interpreted DMRS object. If there is no `dmrs` key in the result, return `None`.

Raises ***InterfaceError*** – when the value is not a dictionary or `delphin.codecs.dmrsjson` is unavailable

eds()

Interpret and return an Eds object.

If `delphin.codecs.eds` is available and the value of the `eds` key in the result is a valid "native" EDS serialization, or if `delphin.codecs.edsjson` is available and the value is a dictionary, return the interpreted EDS object. If there is no `eds` key in the result, return `None`.

Raises ***InterfaceError*** – when the value is an unsupported type or the corresponding module is unavailable

mrs()

Interpret and return an MRS object.

If `delphin.codecs.simplemrs` is available and the value of the `mrs` key in the result is a valid SimpleMRS string, or if `delphin.codecs.mrsjson` is available and the value is a dictionary, return the interpreted MRS object. If there is no `mrs` key in the result, return `None`.

Raises ***InterfaceError*** – when the value is an unsupported type or the corresponding module is unavailable

tree()

Interpret and return a labeled syntax tree.

The tree data may be a standalone datum, or embedded in a derivation.

17.2 Exceptions

exception delphin.interface.**InterfaceError**(**args*, ***kwargs*)
Bases: *delphin.exceptions.PyDelphinException*

Raised on invalid interface operations.

17.3 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 import interface
>>> from delphin import ace
>>> from delphin import repp
>>>
>>> class REPPWrapper(interface.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-2018-x86-64-0.9.30.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():
...         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

EIGHTEEN

DELPHIN.ITSDB

See also:

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

[incr tsdb()] Test Suites

Note: This module implements high-level structures and operations on top of TSDB test suites. For the basic, low-level functionality, see `delphin.tsdb`. For complex queries of the databases, see `delphin.tsql`.

[incr tsdb()] is a tool built on top of TSDB databases for the purpose of profiling and comparing grammar versions using test suites. This module is named after that tool as it also builds higher-level operations on top of TSDB test suites but it has a much narrower scope. The aim of this module is to assist users with creating, processing, or manipulating test suites.

The typical test suite 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
```

18.1 Test Suite Classes

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

- `TestSuite`
- `Table`
- `Row`

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

Bases: `delphin.tsdb.Database`

A [incr tsdb()] test suite database.

Parameters

- **path** – the path to the test suite's directory
- **schema** (*dict*, *str*) – the database schema; either a mapping of table names to lists of `Fields` 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 test suite

schema
database schema as a mapping of table names to lists of `Field` objects

Type dict

encoding
character encoding used when reading and writing tables

Type str

commit()
Commit the current changes to disk.

This method writes the current state of the test suite to disk. The effect is similar to using `tsdb.write_database()`, except that it also updates the test suite's internal bookkeeping so that it is aware that the current transaction is complete. It also may be more efficient if the only changes are adding new rows to existing tables.

property in_transaction
Return `True` is there are uncommitted changes.

property path
The database directory's path.

process(*cpu*, *selector=None*, *source=None*, *fieldmapper=None*, *gzip=False*, *buffer_size=1000*)
Process each item in a [incr tsdb()] test suite.

The output rows will be flushed to disk when the number of new rows in a table is *buffer_size*.

Parameters

- **cpu** (*Processor*) – processor interface (e.g., *ACEParser*)
- **selector** – a pair of (table_name, column_name) that specify the table and column used for processor input (e.g., `('item', 'i-input')`)
- **source** (*Database*) – test suite from which inputs are taken; if `None`, use the current test suite
- **fieldmapper** (*FieldMapper*) – object for mapping response fields to [incr tsdb()] fields; if `None`, use a default mapper for the standard schema
- **gzip** – if `True`, compress non-empty tables with gzip
- **buffer_size** (*int*) – number of output rows to hold in memory before flushing to disk; ignored if the test suite is all in-memory; if `None`, do not flush to disk

Examples

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

processed_items(*fieldmapper=None*)
Iterate over the data as `Response` objects.

reload()
Discard temporary changes and reload the database from disk.

select_from(*name*, *columns=None*, *cast=True*)
Select fields given by *names* from each row in table *name*.

If no field names are given, all fields are returned.

If *cast* is `False`, simple tuples of raw data are returned instead of `Row` objects.

Yields Row

Examples

```
>>> next(ts.select_from('item'))
Row(10, 'unknown', 'formal', 'none', 1, 'S', 'It rained.', ...)
>>> next(ts.select_from('item', ('i-id')))
Row(10)
>>> next(ts.select_from('item', ('i-id', 'i-input')))
Row(10, 'It rained.')
>>> next(ts.select_from('item', ('i-id', 'i-input'), cast=False))
('10', 'It rained.')
```

class `delphin.itsdb.`**`Table`**(*dir*, *name*, *fields*, *encoding='utf-8'*)

Bases: `delphin.tsdb.Relation`

A [incr tsdb()] table.

Parameters

- **`dir`** – path to the database directory
- **`name`** – name of the table
- **`fields`** – the table schema; an iterable of `tsdb.Field` objects
- **`encoding`** – character encoding of the table file

`dir`
The path to the database directory.

`name`
The name of the table.

`fields`
The table's schema.

`encoding`
The character encoding of table files.

`append`(*row*)
Add *row* to the end of the table.

Parameters **`row`** – a `Row` or other iterable containing column values

`clear`()
Clear the table of all rows.

`close`()
Close the table file being iterated over, if open.

`column_index`(*name*)
Return the tuple index of the column with name *name*.

`extend`(*rows*)
Add each row in *rows* to the end of the table.

Parameters **`row`** – an iterable of `Row` or other iterables containing column values

`get_field`(*name*)
Return the `tsdb.Field` object with column name *name*.

select (**names*, *cast=True*)
Select fields given by *names* from each row in the table.

If no field names are given, all fields are returned.

If *cast* is `False`, simple tuples of raw data are returned instead of `Row` objects.

Yields Row

Examples

```
>>> next(table.select())
Row(10, 'unknown', 'formal', 'none', 1, 'S', 'It rained.', ...)
>>> next(table.select('i-id'))
Row(10)
>>> next(table.select('i-id', 'i-input'))
Row(10, 'It rained.')
>>> next(table.select('i-id', 'i-input', cast=False))
('10', 'It rained.')
```

update (*index*, *data*)
Update the row at *index* with *data*.

Parameters

- **index** – the 0-based index of the row in the table
- **data** – a mapping of column names to values for replacement

Examples

```
>>> table.update(0, {'i-input': '...'})
```

class delphin.itsdb.**Row** (*fields*, *data*, *field_index=None*)
A row in a [incr tsdb()] table.

The third argument, *field_index*, is optional. Its purpose is to reduce memory usage because the same field index can be shared by all rows for a table, but using an incompatible index can yield unexpected results for value retrieval by field names (`row[field_name]`).

Parameters

- **fields** – column descriptions; an iterable of `tsdb.Field` objects
- **data** – raw column values
- **field_index** – mapping of field name to its index in *fields*; if not given, it will be computed from *fields*

fields
The fields of the row.

data
The raw column values.

keys ()
Return the list of field names for the row.

Note this returns the names of all fields, not just those with the `:key` flag.

18.2 Processing Test Suites

The *TestSuite.process()* method takes an optional *FieldMapper* object which manages the mapping of data in *Response* objects from a *Processor* to the tables and columns of a test suite. In most cases the user will not need to customize or instantiate these objects as the default works with standard [incr tsdb()] schemas, but *FieldMapper* can be subclassed in order to handle non-standard schemas, e.g., for machine translation workflows.

class delphin.itsdb.**FieldMapper**(*source=None*)

A class for mapping between response objects and test suites.

If *source* is given, it is the test suite providing the inputs used to create the responses, and it is used to provide some contextual information that may not be present in the response.

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

And one method for mapping test suites to responses:

- *collect()* – yield *Response* objects by collecting the relevant data from the test suite

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 rows for, but it may also include those that rely on the previous set (e.g., treebanking preferences, etc.).

Alternative [incr tsdb()] schemas can be handled by overriding these three methods and the __init__() method. Note that overriding *collect()* is only necessary for mapping back from test suites to responses.

affected_tables

list of tables that are affected by the processing

map(*response*)

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

cleanup()

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

collect(*ts*)

Map from test suites to response objects.

The data in the test suite must be ordered.

Note: This method stores the 'item', 'parse', and 'result' tables in memory during operation, so it is not recommended when a test suite is very large as it may exhaust the system's available memory.

18.3 Utility Functions

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

Yields *tuple* –

a triple containing the matched value for *key*, the list of any matching rows from *rows1*, and the list of any matching rows from *rows2*

18.4 Exceptions

exception `delphin.itsdb.`**`ITSDBError`**(**args*, ***kwargs*)

Bases: `delphin.tsdb.TSDBError`

Raised when there is an error processing a [incr tsdb()] profile.

CHAPTER

NINETEEN

DELPHIN.LNK

Surface alignment for semantic entities.

In DELPH-IN semantic representations, entities are aligned to the input surface string is through the so-called "lnk" (pronounced "link") values. There are four types of lnk values which align to the surface in different ways:

- Character spans (also called "characterization pointers"); e.g., `<0:4>`
- Token indices; e.g., `<0 1 3>`
- Chart vertex spans; e.g., `<0#2>`
- Edge identifier; e.g., `<@42>`

The latter two are unlikely to be encountered by users. Chart vertices were used by the PET parser but are now essentially deprecated and edge identifiers are only used internally in the LKB for generation. I will therefore focus on the first two kinds.

Character spans (sometimes called "characterization pointers") are by far the most commonly used type—possibly even the only type most users will encounter. These spans indicate the positions *between* characters in the input string that correspond to a semantic entity, similar to how Python and Perl do string indexing. For example, `<0:4>` would capture the first through fourth characters—a span that would correspond to the first word in a sentence like "Dogs bark". These spans assume the input is a flat, or linear, string and can only select contiguous chunks. Character spans are used by REPP (the Regular Expression PreProcessor; see `delphin.repp`) to track the surface alignment prior to string changes introduced by tokenization.

Token indices select input tokens rather than characters. This method, though not widely used, is more suitable for input sources that are not flat strings (e.g., a lattice of automatic speech recognition (ASR) hypotheses), or where non-contiguous sequences are needed (e.g., from input containing markup or other noise).

Note: Much of this background is from comments in the LKB source code: See: http://svn.emmtee.net/trunk/lingo/lkb/src/mrs/lnk.lisp

---

Support for lnk values in PyDelphin is rather simple. The `Lnk` class is able to parse lnk strings and model the contents for serialization of semantic representations. In addition, semantic entities such as DMRS `Nodes` and MRS `EPs` have `cfrom` and `cto` attributes which are the start and end pointers for character spans (defaulting to `-1` if a character span is not specified for the entity).

19.1 Classes

class `delphin.lnk.Lnk`(*arg*, *data=None*)

Surface-alignment information for predications.

Lnk objects link predicates to the surface form in one of several ways, the most common of which being the character span of the original string.

Valid types and their associated *data* shown in the table below.

type	data	example
Lnk.CHARSPAN	surface string span	(0, 5)
Lnk.CHARTSPAN	chart vertex span	(0, 5)
Lnk.TOKENS	token identifiers	(0, 1, 2)
Lnk.EDGE	edge identifier	1

Parameters

- **`arg`** – Lnk type or the string representation of a Lnk
- **`data`** – alignment data (assumes *arg* is a Lnk type)

`type`
the way the Lnk relates the semantics to the surface form

`data`
the alignment data (depends on the Lnk type)

Example

```
>>> Lnk('<0:5>').data
(0, 5)
>>> str(Lnk.charspan(0,5))
'<0:5>'
>>> str(Lnk.chartspan(0,5))
'<0#5>'
>>> str(Lnk.tokens([0,1,2]))
'<0 1 2>'
>>> str(Lnk.edge(1))
'<@1>'
```

`classmethod charspan`(*start, end*)
Create a Lnk object for a character span.

Parameters

- **`start`** – the initial character position (cfrom)
- **`end`** – the final character position (cto)

`classmethod chartspan`(*start, end*)
Create a Lnk object for a chart span.

Parameters

- **`start`** – the initial chart vertex
- **`end`** – the final chart vertex

`classmethod default`()
Create a Lnk object for when no information is given.

`classmethod edge`(*edge*)
Create a Lnk object for an edge (used internally in generation).

Parameters **`edge`** – an edge identifier

classmethod tokens(*tokens*)
Create a Lnk object for a token range.

Parameters **`tokens`** – a list of token identifiers

class `delphin.lnk.`**`LnkMixin`**(*lnk=None*, *surface=None*)
A mixin class for adding `cfrom` and `cto` properties on structures.

property cfrom
The initial character position in the surface string.

Defaults to -1 if there is no valid cfrom value.

property cto
The final character position in the surface string.

Defaults to -1 if there is no valid cto value.

19.2 Exceptions

exception `delphin.lnk.`**`LnkError`**(**args*, ***kwargs*)
Bases: *`delphin.exceptions.PyDelphinException`*

Raised on invalid Lnk values or operations.

CHAPTER

TWENTY

DELPHIN.PREDICATE

Semantic predicates.

Semantic predicates are atomic symbols representing semantic entities or constructions. For example, in the English Resource Grammar, `_mouse_n_1` is the predicate for the word *mouse*, but it is underspecified for lexical semantics—it could be an animal, a computer's pointing device, or something else. Another example from the ERG is `compound`, which is used to link two compounded nouns, such as for *mouse pad*.

There are two main categories of predicates: **abstract** and **surface**. In form, abstract predicates do not begin with an underscore and in usage they often correspond to semantic constructions that are not represented by a token in the input, such as the `compound` example above. Surface predicates, in contrast, are the semantic representation of surface (i.e., lexical) tokens, such as the `_mouse_n_1` example above. In form, they must always begin with a single underscore, and have two or three components: lemma, part-of-speech, and (optionally) sense.

See also:

- The DELPH-IN wiki about predicates: http://moin.delph-in.net/PredicateRfc

In DELPH-IN there is the concept of "real predicates" which are surface predicates decomposed into their lemma, part-of-speech, and sense, but in PyDelphin (as of v1.0.0) predicates are always simple strings. However, this module has functions for composing and decomposing predicates from/to their components (the `create()` and `split()` functions, respectively). In addition, there are functions to normalize (`normalize()`) and validate (`is_valid()`, `is_surface()`, `is_abstract()`) predicate symbols.

20.1 Module Functions

`delphin.predicate.split(s)`

Split predicate string *s* and return the lemma, pos, and sense.

This function uses more robust pattern matching than used by the validation functions `is_valid()`, `is_surface()`, and `is_abstract()`. This robustness is to accommodate inputs that are not entirely well-formed, such as surface predicates with underscores in the lemma or a missing part-of-speech. Additionally it can be used, with some discretion, to inspect abstract predicates, which technically do not have individual components but in practice follow the same convention as surface predicates.

Examples

```
>>> split('_dog_n_1_rel')
('dog', 'n', '1')
>>> split('udef_q')
('udef', 'q', None)
```

delphin.predicate.**create**(*lemma*, *pos*, *sense=None*)

Create a surface predicate string from its *lemma*, *pos*, and *sense*.

The components are validated in order to guarantee that the resulting predicate symbol is well-formed.

This function cannot be used to create abstract predicate symbols.

Examples

```
>>> create('dog', 'n', '1')
'_dog_n_1'
>>> create('some', 'q')
'_some_q'
```

delphin.predicate.**normalize**(*s*)

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

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

Examples

```
>>> normalize('"_DOG_n_1_rel"')
'_dog_n_1'
>>> normalize('_dog_n_1')
'_dog_n_1'
```

delphin.predicate.**is_valid**(*s*)

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

Examples

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

delphin.predicate.**is_surface**(*s*)

Return `True` if *s* is a valid surface predicate string.

Examples

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

delphin.predicate.**is_abstract**(*s*)

Return `True` if *s* is a valid abstract predicate string.

Examples

```
>>> is_abstract('udef_q_rel')
True
>>> is_abstract('"coord"')
True
>>> is_valid('"_dog_n_1_rel"')
False
>>> is_valid('_dog_n_1')
False
```

20.2 Exceptions

exception delphin.predicate.**PredicateError**(**args*, ***kwargs*)

Bases: *delphin.exceptions.PyDelphinException*

Raised on invalid predicate or predicate operations.

CHAPTER

TWENTYONE

DELPHIN.MRS

Minimal Recursion Semantics ([MRS]).

21.1 Serialization Formats

21.2 Module Constants

delphin.mrs.**INTRINSIC_ROLE**
The `ARG0` role that is associated with the intrinsic variable (*EP.iv*).

delphin.mrs.**RESTRICTION_ROLE**
The `RSTR` role used to select the restriction of a quantifier.

delphin.mrs.**BODY_ROLE**
The `BODY` role used to select the body of a quantifier.

delphin.mrs.**CONSTANT_ROLE**
The `CARG` role used to encode the constant value (*EP.carg*) associated with certain kinds of predications, such as named entities, numbers, etc.

21.3 Classes

class delphin.mrs.**MRS**(*top=None, index=None, rels=None, hcons=None, icons=None, variables=None, lnk=None, surface=None, identifier=None*)
Bases: *delphin.scope.ScopingSemanticStructure*

A semantic representation in Minimal Recursion Semantics.

Parameters

- **top** – the top scope handle
- **index** – the top variable
- **rels** – iterable of EP relations
- **hcons** – iterable of handle constraints
- **icons** – iterable of individual constraints
- **variables** – mapping of variables to property maps
- **lnk** – surface alignment
- **surface** – surface string

- **identifier** – a discourse-utterance identifier

top
The top scope handle.

index
The top variable.

rels
The list of EPs (alias of *predications*).

hcons
The list of handle constraints.

icons
The list of individual constraints.

variables
A mapping of variables to property maps.

lnk
The surface alignment for the whole MRS.

surface
The surface string represented by the MRS.

identifier
A discourse-utterance identifier.

arguments(*types=None*, *expressed=None*)
Return a mapping of the argument structure.

Parameters

- **types** – an iterable of predication types to include
- **expressed** – if `True`, only include arguments to expressed predications; if `False`, only include those unexpressed; if `None`, include both

Returns A mapping of predication ids to lists of (role, target) pairs for outgoing arguments for the predication.

is_quantifier(*id*)
Return `True` if *var* is the bound variable of a quantifier.

properties(*id*)
Return the properties associated with EP *id*.

Note that this function returns properties associated with the intrinsic variable of the EP whose id is *id*. To get the properties of a variable directly, use *variables*.

quantification_pairs()
Return a list of (Quantifiee, Quantifier) pairs.

Both the Quantifier and Quantifiee are `Predication` objects, unless they do not quantify or are not quantified by anything, in which case they are `None`. In well-formed and complete structures, the quantifiee will never be `None`.

Example

```
>>> [(p.predicate, q.predicate)
...   for p, q in m.quantification_pairs()]
[('_dog_n_1', '_the_q'), ('_bark_v_1', None)]
```

scopal_arguments(*scopes=None*)

Return a mapping of the scopal argument structure.

Unlike `SemanticStructure.arguments()`, the list of arguments is a 3-tuple including the scopal relation: (role, scope_relation, scope_label).

Parameters **scopes** – mapping of scope labels to lists of predications

scopes()

Return a tuple containing the top label and the scope map.

Note that the top label is different from *top*, which is the handle that is qeq to the top scope's label. If *top* does not select a top scope, the `None` is returned for the top label.

The scope map is a dictionary mapping scope labels to the lists of predications sharing a scope.

class delphin.mrs.**EP**(*predicate*, *label*, *args=None*, *lnk=None*, *surface=None*, *base=None*)

Bases: *delphin.sembase.Predication*

An MRS elementary predication (EP).

EPs combine a predicate with various structural semantic properties. They must have a `predicate`, and `label`. Arguments are optional. Intrinsic arguments (`ARG0`) are not strictly required, but they are important for many semantic operations, and therefore it is a good idea to include them.

Parameters

- **predicate** – semantic predicate
- **label** – scope handle
- **args** – mapping of roles to values
- **lnk** – surface alignment
- **surface** – surface string
- **base** – base form

id

an identifier (same as *iv* except for quantifiers which replace the `iv`'s variable type with `q`)

predicate

semantic predicate

label

scope handle

args

mapping of roles to values

iv

intrinsic variable (shortcut for `args['ARG0']`)

carg

constant argument (shortcut for `args['CARG']`)

lnk

surface alignment

cfrom
surface alignment starting position

Type int

cto
surface alignment ending position

Type int

surface
surface string

base
base form

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

class delphin.mrs.**HCons**
A relation between two handles.

Parameters

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

property hi
The higher-scoped handle.

property lo
The lower-scoped handle.

property relation
The constraint relation.

class delphin.mrs.**ICons**
Individual Constraint: A relation between two variables.

Parameters

- **left** – intrinsic variable of the constraining EP
- **relation** – relation of the constraint
- **right** – intrinsic variable of the constrained EP

property left
The intrinsic variable of the constraining EP.

property relation
The constraint relation.

property right
The intrinsic variable of the constrained EP.

21.4 Module Functions

`delphin.mrs.`**`is_connected`**`(`*m*`)`
Return `True` if *m* is a fully-connected MRS.

A connected MRS is one where, when viewed as a graph, all EPs are connected to each other via regular (non-scopal) arguments, scopal arguments (including qeqs), or label equalities.

`delphin.mrs.`**`has_intrinsic_variable_property`**`(`*m*`)`
Return `True` if *m* satisfies the intrinsic variable property.

An MRS has the intrinsic variable property when it passes the following:

- *`has_complete_intrinsic_variables()`*
- *`has_unique_intrinsic_variables()`*

Note that for quantifier EPs, `ARG0` is overloaded to mean "bound variable". Each quantifier should have an `ARG0` that is the intrinsic variable of exactly one non-quantifier EP, but this function does not check for that.

`delphin.mrs.`**`has_complete_intrinsic_variables`**`(`*m*`)`
Return `True` if all non-quantifier EPs have intrinsic variables.

`delphin.mrs.`**`has_unique_intrinsic_variables`**`(`*m*`)`
Return `True` if all intrinsic variables are unique to their EPs.

`delphin.mrs.`**`is_well_formed`**`(`*m*`)`
Return `True` if MRS *m* is well-formed.

A well-formed MRS meets the following criteria:

- *`is_connected()`*
- *`has_intrinsic_variable_property()`*
- *`plausibly_scopes()`*

The final criterion is a heuristic for determining if the MRS scopes by checking if handle constraints and scopal arguments have any immediate violations (e.g., a scopal argument selecting the label of its EP).

`delphin.mrs.`**`plausibly_scopes`**`(`*m*`)`
Quickly test if MRS *m* can plausibly resolve a scopal reading.

This tests a number of things:

- Is the MRS's top qeq to a label
- Do any EPs scope over themselves
- Do multiple EPs use the handle constraint
- Is the lo handle of a qeq not actually a label
- Are any qeqs not selected by an EP

It does not test for transitive scopal plausibility.

`delphin.mrs.`**`is_isomorphic`**`(`*m1*, *m2*, *properties=True*`)`
Return `True` if *m1* and *m2* are isomorphic MRSs.

Isomorphicity compares the predicates of a semantic structure, the morphosemantic properties of their predications (if `properties=True`), constant arguments, and the argument structure between predications. Non-semantic properties like identifiers and surface alignments are ignored.

Parameters

- **m1** – the left MRS to compare
- **m2** – the right MRS to compare
- **properties** – if `True`, ensure variable properties are equal for mapped predications

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

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

Parameters

- **testbag** – An iterable of MRS objects to test
- **goldbag** – An iterable of MRS objects to compare against
- **properties** – if `True`, ensure variable properties are equal for mapped predications
- **count_only** – If `True`, the returned triple will only have the counts of each; if `False`, a list of MRS 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 MRS objects otherwise.

`delphin.mrs.`**`from_dmrs`**(*d*)

Create an MRS by converting from DMRS *d*.

Parameters d – the input DMRS

Returns MRS

Raises MRSError when conversion fails. –

21.5 Exceptions

exception `delphin.mrs.`**`MRSError`**(**args*, ***kwargs*)

Bases: *delphin.exceptions.PyDelphinException*

Raises on invalid MRS operations.

exception `delphin.mrs.`**`MRSSyntaxError`**(*message=None*, *filename=None*, *lineno=None*, *offset=None*, *text=None*)

Bases: *delphin.exceptions.PyDelphinSyntaxError*

Raised when an invalid MRS serialization is encountered.

CHAPTER

TWENTYTWO

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.

Note: Requires `regex` (https://bitbucket.org/mrabarnett/mrab-regex/), for advanced regular expression features such as group-local inline flags. Without it, PyDelphin will fall back to the `re` module in the standard library which may give some unexpected results. The `regex` library, however, will not parse unescaped brackets in character classes without resorting to a compatibility mode (see this issue for the ERG), and PyDelphin will warn if this happens. The `regex` dependency is satisfied if you install PyDelphin with the `[repp]` extra (see *Requirements, Installation, and Testing*).

22.1 Module Constants

delphin.repp.**DEFAULT_TOKENIZER = '[\\t]+'**

The tokenization pattern used if none is given in a REPP module.

22.2 Classes

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

tokenize_result (*result, pattern='[\\t]+'*)
Tokenize the result of rule application.

Parameters

- **result** – a *REPPResult* object
- **pattern** (*str, optional*) – the regular expression pattern on which to split tokens; defaults to `[ ]+`

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** (*string, startmap, endmap*)
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`

property endmap
Alias for field number 2

property startmap
Alias for field number 1

property string
Alias for field number 0

class delphin.repp.**REPPStep**(*input*, *output*, *operation*, *applied*, *startmap*, *endmap*)
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`

property applied
Alias for field number 3

property endmap
Alias for field number 5

property input
Alias for field number 0

property operation
Alias for field number 2

property output
Alias for field number 1

property startmap
Alias for field number 4

22.3 Exceptions

exception delphin.repp.**REPPError**(**args*, ***kwargs*)
Bases: *delphin.exceptions.PyDelphinException*

Raised when there is an error in tokenizing with REPP.

exception delphin.repp.**REPPWarning**(**args*, ***kwargs*)
Bases: *delphin.exceptions.PyDelphinWarning*

Issued when REPP may not behave as expected.

CHAPTER

TWENTYTHREE

DELPHIN.SEMBASE

Basic classes and functions for semantic representations.

23.1 Module Functions

delphin.sembase.**role_priority**(*role*)
Return a representation of role priority for ordering.

delphin.sembase.**property_priority**(*prop*)
Return a representation of property priority for ordering.

Note: The ordering provided by this function was modeled on the ERG and Jacy grammars and may be inaccurate for others. Properties not known to this function will be sorted alphabetically.

23.2 Classes

class delphin.sembase.**Predication**(*id*, *predicate*, *type*, *lnk*, *surface*, *base*)
Bases: *delphin.lnk.LnkMixin*

An instance of a predicate in a semantic structure.

While a predicate (see *delphin.predicate*) is a description of a possible semantic entity, a predication is the instantiation of a predicate in a semantic structure. Thus, multiple predicates with the same form are considered the same thing, but multiple predications with the same predicate will have different identifiers and, if specified, different surface alignments.

class delphin.sembase.**SemanticStructure**(*top*, *predications*, *lnk*, *surface*, *identifier*)
Bases: *delphin.lnk.LnkMixin*

A basic semantic structure.

DELPH-IN-style semantic structures are rooted DAGs with flat lists of predications.

Parameters

- **top** – identifier for the top of the structure
- **predications** – list of predications in the structure
- **identifier** – a discourse-utterance identifier

top
identifier for the top of the structure

predications
: list of predications in the structure

identifier
: a discourse-utterance identifier

arguments(*types=None*, *expressed=None*)
: Return a mapping of the argument structure.

Parameters

- **types** – an iterable of predication types to include
- **expressed** – if `True`, only include arguments to expressed predications; if `False`, only include those unexpressed; if `None`, include both

Returns A mapping of predication ids to lists of (role, target) pairs for outgoing arguments for the predication.

is_quantifier(*id*)
: Return `True` if *id* represents a quantifier.

properties(*id*)
: Return the morphosemantic properties for *id*.

quantification_pairs()
: Return a list of (Quantifiee, Quantifier) pairs.

Both the Quantifier and Quantifiee are *Predication* objects, unless they do not quantify or are not quantified by anything, in which case they are `None`. In well-formed and complete structures, the quantifiee will never be `None`.

Example

```
>>> [(p.predicate, q.predicate)
...  for p, q in m.quantification_pairs()]
[('_dog_n_1', '_the_q'), ('_bark_v_1', None)]
```

CHAPTER

TWENTYFOUR

DELPHIN.SCOPE

Structures and operations for quantifier scope in DELPH-IN semantics.

While the predicate-argument structure of a semantic representation is a directed-acyclic graph, the quantifier scope is a tree overlayed on the edges of that graph. In a fully scope-resolved structure, there is one tree spanning the entire graph, but in underspecified representations like MRS, there are multiple subtrees that span the graph nodes but are not all connected together. The components are then connected via qeq constraints which specify a partial ordering for the tree such that quantifiers may float in between the nodes connected by qeqs.

Each node in the scope tree (called a *scopal position*) may encompass multiple nodes in the predicate-argument graph. Nodes that share a scopal position are said to be in a *conjunction*.

The dependency representations EDS and DMRS develop the idea of scope representatives (called *representative nodes* or sometimes *heads*), whereby a single node is selected from a conjunction to represent the conjunction as a whole.

24.1 Classes

class `delphin.scope.`**`ScopingSemanticStructure`**(*top*, *index*, *predications*, *lnk*, *surface*, *identifier*)

Bases: `delphin.sembase.SemanticStructure`

A semantic structure that encodes quantifier scope.

This is a base class for semantic representations, namely `MRS` and `DMRS`, that distinguish scopal and non-scopal arguments. In addition to the attributes and methods of the `SemanticStructure` class, it also includes an `index` which indicates the non-scopal top of the structure, `scopes()` for describing the labeled scopes of a structure, and `scopal_arguments()` for describing the arguments that select scopes.

`index`

The non-scopal top of the structure.

`scopal_arguments`(*scopes=None*)

Return a mapping of the scopal argument structure.

Unlike `SemanticStructure.arguments()`, the list of arguments is a 3-tuple including the scopal relation: (role, scope_relation, scope_label).

Parameters **`scopes`** – mapping of scope labels to lists of predications

`scopes`()

Return a tuple containing the top label and the scope map.

The top label is the label of the top scope in the scope map.

The scope map is a dictionary mapping scope labels to the lists of predications sharing a scope.

24.2 Module Functions

`delphin.scope.`**`conjoin`**(*scopes*, *leqs*)

Conjoin multiple scopes with equality constraints.

Parameters

- **scopes** – a mapping of scope labels to predications
- **leqs** – a list of pairs of equated scope labels

Returns A mapping of the labels to the predications of each conjoined scope. The conjoined scope labels are taken arbitrarily from each equated set).

Example

```
>>> conjoined = scope.conjoin(mrs.scopes(), [('h2', 'h3')])
>>> {lbl: [p.id for p in ps] for lbl, ps in conjoined.items()}
{'h1': ['e2'], 'h2': ['x4', 'e6']}
```

`delphin.scope.`**`descendants`**(*x*, *scopes=None*)

Return a mapping of predication ids to their scopal descendants.

Parameters

- **x** – an MRS or a DMRS
- **scopes** – a mapping of scope labels to predications

Returns A mapping of predication ids to lists of predications that are scopal descendants.

Example

```
>>> m = mrs.MRS(...)  # Kim didn't think that Sandy left.
>>> descendants = scope.descendants(m)
>>> for id, ds in descendants.items():
...     print(m[id].predicate, [d.predicate for d in ds])
...
proper_q ['named']
named []
neg ['_think_v_1', '_leave_v_1']
_think_v_1 ['_leave_v_1']
_leave_v_1 []
proper_q ['named']
named []
```

`delphin.scope.`**`representatives`**(*x*, *priority=None*)

Find the scope representatives in *x* sorted by *priority*.

When predications share a scope, generally one takes another as a non-scopal argument. For instance, the ERG analysis of a phrase like "very old book" has the predicates `_very_x_deg`, `_old_a_1`, and `_book_n_of` which all share a scope, where `_very_x_deg` takes `_old_a_1` as its `ARG1` and `_old_a_1` takes `_book_n_of` as its `ARG1`. Predications that do not take any other predication within their scope as an argument (as `_book_n_of` above does not) are scope representatives.

priority is a function that takes a `Predication` object and returns a rank which is used to to sort the representatives for each scope. As the predication alone might not contain enough information for useful sorting, it can be helpful to create a function configured for the input semantic structure *x*. If *priority* is `None`, representatives are sorted according to the following criteria:

1. Prefer predications that are quantifiers or instances (type 'x')
2. Prefer eventualities (type 'e') over other types
3. Prefer tensed over untensed eventualities
4. Finally, prefer prefer those appearing first in *x*

The definition of "tensed" vs "untensed" eventualities is grammar-specific, but it is used by several large grammars. If a grammar does something different, criterion (3) is ignored. Criterion (4) is not linguistically motivated but is used as a final disambiguator to ensure consistent results.

Parameters

- **x** – an MRS or a DMRS
- **priority** – a function that maps an EP to a rank for sorting

Example

```
>>> sent = 'The new chef whose soup accidentally spilled quit.'
>>> m = ace.parse(erg, sent).result(0).mrs()
>>> # in this example there are 4 EPs in scope h7
>>> _, scopes = m.scopes()
>>> [ep.predicate for ep in scopes['h7']]
['_new_a_1', '_chef_n_1', '_accidental_a_1', '_spill_v_1']
>>> # there are 2 representatives for scope h7
>>> reps = scope.representatives(m)['h7']
>>> [ep.predicate for ep in reps]
['_chef_n_1', '_spill_v_1']
```

24.3 Exceptions

exception delphin.scope.**ScopeError**(**args*, ***kwargs*)

Bases: *delphin.exceptions.PyDelphinException*

Raised on invalid scope operations.

CHAPTER

TWENTYFIVE

DELPHIN.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:

The following DELPH-IN wikis contain more information:

- Technical specifications: http://moin.delph-in.net/SemiRfc
- Overview and usage: http://moin.delph-in.net/RmrsSemi

25.1 Loading a SEM-I from a File

The `load()` module function is used to read the regular file-based SEM-I definitions, but there is also a dictionary representation which may be useful for JSON serialization, e.g., for an HTTP API that makes use of SEM-Is. See `SemI.to_dict()` for the later.

`delphin.semi.load`(*source*, *encoding='utf-8'*)
Interpret and return the SEM-I defined at path *source*.

Parameters

- **source** – the path of the top file for the SEM-I. Note: this must be a path and not an open file.
- **encoding** (`str`) – the character encoding of the file

Returns The SemI defined by *source*

25.2 The SemI Class

The main class modeling a semantic interface is `SemI`. The predicate synopses have enough complexity that two more subclasses are used to make inspection easier: `Synopsis` contains the role information for an individual predicate synopsis, and each role is modeled with a `SynopsisRole` class.

class `delphin.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, {'parents': [. . .], 'properties': [. . .]})
- **properties** – a mapping of (prop, {'parents': [. . .]})
- **roles** – a mapping of (role, {'value': . . . })
- **predicates** – a mapping of (pred, {'parents': [. . .], 'synopses': [. . .]})

variables
a *MultiHierarchy* of variables; node data contains the property lists

properties
a *MultiHierarchy* of properties

roles
mapping of role names to allowed variable types

predicates
a *MultiHierarchy* of predicates; node data contains lists of synopses

The data in the SEM-I can be directly inspected via the *variables*, *properties*, *roles*, and *predicates* attributes.

```
>>> smi = semi.load('../grammars/erg/etc/erg.smi')
>>> smi.variables['e']
<delphin.tfs.TypeHierarchyNode object at 0x7fa02f877388>
>>> smi.variables['e'].parents
['i']
>>> smi.variables['e'].data
[('SF', 'sf'), ('TENSE', 'tense'), ('MOOD', 'mood'), ('PROG', 'bool'), ('PERF',
↪'bool')]
>>> 'sf' in smi.properties
True
>>> smi.roles['ARG0']
'i'
>>> for synopsis in smi.predicates['can_able'].data:
...     print(', '.join('{0.name} {0.value}'.format(roledata)
...                     for roledata in synopsis))
...
ARG0 e, ARG1 i, ARG2 p
>>> smi.predicates.descendants('some_q')
['_another_q', '_many+a_q', '_an+additional_q', '_what+a_q', '_such+a_q', '_some_
↪q_indiv', '_some_q', '_a_q']
```

Note that the variables, properties, and predicates are *TypeHierarchy* objects.

find_synopsis (*predicate*, *args=None*)
Return the first matching synopsis for *predicate*.

predicate will be normalized before lookup.

Synopses can be matched by a description of arguments which is tested with *Synopsis.subsumes()*. If no condition is given, the first synopsis is returned.

Parameters

- **predicate** – predicate symbol whose synopsis will be returned
- **args** – description of arguments that must be subsumable by the synopsis

Returns matching synopsis as a list of `(role, value, properties, optional)` role tuples

Raises *SemIError* – if *predicate* is undefined or if no matching synopsis can be found

Example

```
>>> smi.find_synopsis('_write_v_to')
[('ARG0', 'e', [], False), ('ARG1', 'i', [], False),
 ('ARG2', 'p', [], True), ('ARG3', 'h', [], True)]
>>> smi.find_synopsis('_write_v_to', args='eii')
[('ARG0', 'e', [], False), ('ARG1', 'i', [], False),
 ('ARG2', 'i', [], False)]
```

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

to_dict ()
Return a dictionary representation of the SemI.

class delphin.semi.**Synopsis** (*roles*)
A SEM-I predicate synopsis.

A synopsis describes the roles of a predicate in a semantic structure, so it is no more than a tuple of roles as *SynopsisRole* objects. The length of the synopsis is thus the arity of a predicate while the individual role items detail the role names, argument types, associated properties, and optionality.

classmethod from_dict (*d*)
Create a Synopsis from its dictionary representation.

Example:

```
>>> synopsis = Synopsis.from_dict({
...     'roles': [
...         {'name': 'ARG0', 'value': 'e'},
...         {'name': 'ARG1', 'value': 'x',
...          'properties': {'NUM': 'sg'}}
...     ]
... })
...
>>> len(synopsis)
2
```

subsumes (*args*, *variables=None*)
Return `True` if the Synopsis subsumes *args*.

The *args* argument is a description of MRS arguments. It may take two different forms:

- a sequence (e.g., string or list) of variable types, e.g., `"exh"`, which must be subsumed by the role values of the synopsis in order
- a mapping (e.g., a dict) of roles to variable types which must match roles in the synopsis; the variable type may be `None` which matches any role value

In both cases, the sequence or mapping must be a subset of the roles of the synopsis, and any missing must be optional roles, otherwise the synopsis does not subsume *args*.

The *variables* argument is a variable hierarchy. If it is `None`, variables will be checked for strict equality.

to_dict ()
Return a dictionary representation of the Synopsis.

Example:

```
>>> Synopsis([
...     SynopsisRole('ARG0', 'e'),
...     SynopsisRole('ARG1', 'x', {'NUM': 'sg'})
... ]).to_dict()
{'roles': [{'name': 'ARG0', 'value': 'e'},
           {'name': 'ARG1', 'value': 'x',
            'properties': {'NUM': 'sg'}}]}
```

class delphin.semi.**SynopsisRole**(*name*, *value*, *properties=None*, *optional=False*)

Role data associated with a SEM-I predicate synopsis.

Parameters

- **name** (*str*) – the role name
- **value** (*str*) – the role value (variable type or "string")
- **properties** (*dict*) – properties associated with the role's value
- **optional** (*bool*) – a flag indicating if the role is optional

Example:

```
>>> role = SynopsisRole('ARG0', 'x', {'PERS': '3'}, False)
```

25.3 Exceptions and Warnings

exception delphin.semi.**SemIError**(**args*, ***kwargs*)

Bases: *delphin.exceptions.PyDelphinException*

Raised when loading an invalid SEM-I.

exception delphin.semi.**SemISyntaxError**(*message=None*, *filename=None*, *lineno=None*, *offset=None*, *text=None*)

Bases: *delphin.exceptions.PyDelphinSyntaxError*

Raised when loading an invalid SEM-I.

exception delphin.semi.**SemIWarning**(**args*, ***kwargs*)

Bases: *delphin.exceptions.PyDelphinWarning*

Warning class for questionable SEM-Is.

CHAPTER

TWENTYSIX

DELPHIN.TDL

Contents

Classes and functions for parsing and inspecting TDL.

Type Description Language (TDL) is a declarative language for describing type systems, mainly for the creation of DELPH-IN HPSG grammars. 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 this publication has become outdated to the current usage of TDL in DELPH-IN grammars and its TDL syntax description is inaccurate in places. It is, however, 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.

Below is an example of a basic type from the English Resource Grammar (ERG):

```
basic_word := word_or_infl_rule & word_or_punct_rule &
  [ SYNSEM [ PHON.ONSET.--TL #tl,
             LKEYS.KEYREL [ CFROM #from,
                            CTO #to ] ],
    ORTH [ CLASS #class, FROM #from, TO #to, FORM #form ],
    TOKENS [ +LIST #tl & < [ +CLASS #class, +FROM #from, +FORM #form ], ... >,
             +LAST.+TO #to ] ].
```

The `delphin.tdl` 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.

26.1 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

26.2 Functions

`delphin.tdl.`**`iterparse`**`(`*path*, *encoding='utf-8'*`)`
Parse the TDL file at *path* and iteratively yield parse events.

Parse events are `(event, object, lineno)` tuples, where `event` is a string ("`TypeDefinition`", "`TypeAddendum`", "`LexicalRuleDefinition`", "`LetterSet`", "`WildCard`", "`BeginEnvironment`", "`EndEnvironment`", "`FileInclude`", "`LineComment`", or "`BlockComment`"), `object` is the interpreted TDL object, and `lineno` is the line number where the entity began in *path*.

Parameters

- **`path`** – path to a TDL file
- **`encoding`** (*`str`*) – the encoding of the file (default: "`utf-8`")

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 < > ].
```

26.3 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.

26.3.1 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
coreferncе identifier or tag

Type str

docstring
documentation string

Type str

26.3.2 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`.

property terms
The list of terms in the conjunction.

types()
Return the list of type terms in the conjunction.

26.3.3 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.

property 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

26.3.4 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

26.3.5 Environments and File Inclusion

class `delphin.tdl.`**`TypeEnvironment`**(*entries=None*)
TDL type environment.

Parameters **entries** (*list*) – TDL entries

class `delphin.tdl.`**`InstanceEnvironment`**(*status*, *entries=None*)
TDL instance environment.

Parameters

- **status** (*str*) – status (e.g., "`lex-rule`")
- **entries** (*list*) – TDL entries

class `delphin.tdl.`**`FileInclude`**(*value="*, *basedir="*)
Include other TDL files in the current environment.

Parameters

- **value** – quoted value of the TDL include statement
- **basedir** – directory containing the file with the include statement

value
The quoted value of TDL include statement.

path
The path to the TDL file to include.

26.4 Exceptions and Warnings

exception delphin.tdl.**TDLError**(**args*, ***kwargs*)

Bases: `delphin.exceptions.PyDelphinException`

Raised when there is an error in processing TDL.

exception delphin.tdl.**TDLSyntaxError**(*message=None*, *filename=None*, *lineno=None*, *offset=None*, *text=None*)

Bases: `delphin.exceptions.PyDelphinSyntaxError`

Raised when parsing TDL text fails.

exception delphin.tdl.**TDLWarning**(**args*, ***kwargs*)

Bases: `delphin.exceptions.PyDelphinWarning`

Raised when parsing unsupported TDL features.

CHAPTER

TWENTYSEVEN

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.

27.1 Classes

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.**TypedFeatureStructure**(*type, featvals=None*)

Bases: *delphin.tfs.FeatureStructure*

A typed *FeatureStructure*.

Parameters

- **type** (*str*) – type name

- **featvals** (*dict, list*) – a mapping or iterable of feature paths to feature values

property type
The type assigned to the feature structure.

class delphin.tfs.**TypeHierarchy**(*top*, *hierarchy=None*, *data=None*, *normalize_identifier=None*)
Bases: *delphin.hierarchy.MultiHierarchy*

A Type Hierarchy.

Type hierarchies have certain properties, such as a unique top node, multiple inheritance, case insensitivity, and unique greatest-lower-bound (glb) types.

Note: Checks for unique glbs is not yet implemented.

TypeHierarchies may be constructed when instantiating the class or via the *update()* method using a dictionary mapping type names to node values, or one-by-one using dictionary-like access. In both cases, the node values may be an individual parent name, an iterable of parent names, or a `TypeHierarchyNode` object. Retrieving a node via dictionary access on the typename returns a `TypeHierarchyNode` regardless of the method used to create the node.

```
>>> th = TypeHierarchy('*top*', {'can-fly': '*top*'})
>>> th.update({'can-swim': '*top*', 'can-walk': '*top*'})
>>> th['butterfly'] = ('can-fly', 'can-walk')
>>> th['duck'] = TypeHierarchyNode(
...     ('can-fly', 'can-swim', 'can-walk'),
...     data='some info relating to ducks...')
>>> th['butterfly'].data = 'some info relating to butterflies'
```

In some ways the TypeHierarchy behaves like a dictionary, but it is not a subclass of `dict` and does not implement all its methods. Also note that some methods ignore the top node, which make certain actions easier:

```
>>> th = TypeHierarchy('*top*', {'a': '*top*', 'b': 'a', 'c': 'a'})
>>> len(th)
3
>>> list(th)
['a', 'b', 'c']
>>> TypeHierarchy('*top*', dict(th.items())) == th
True
```

But others do not ignore the top node, namely those where you can request it specifically:

```
>>> '*top*' in th
True
>>> th['*top*']
<TypeHierarchyNode ... >
```

Parameters

- **top** (*str*) – unique top type
- **hierarchy** (*dict*) – mapping of `{child: node}` (see description above concerning the `node` values)

top
the hierarchy's top type

ancestors (*identifier*)
Return the ancestors of *identifier*.

children (*identifier*)
Return the immediate children of *identifier*.

compatible (*a*, *b*)
Return `True` if node *a* is compatible with node *b*.

In a multiply-inheriting hierarchy, node compatibility means that two nodes share a common descendant. It is a commutative operation, so `compatible(a, b) == compatible(b, a)`. Note that in a singly-inheriting hierarchy, two nodes are never compatible by this metric.

Parameters

- **a** – a node identifier
- **b** – a node identifier

Examples

```
>>> h = MultiHierarchy('*top*', {'a': '*top*',
...                              'b': '*top*'})
>>> h.compatible('a', 'b')
False
>>> h.update({'c': 'a b'})
>>> h.compatible('a', 'b')
True
```

descendants (*identifier*)
Return the descendants of *identifier*.

items ()
Return the (identifier, data) pairs excluding the top node.

parents (*identifier*)
Return the immediate parents of *identifier*.

subsumes (*a*, *b*)
Return `True` if node *a* subsumes node *b*.

A node is subsumed by the other if it is a descendant of the other node or if it is the other node. It is not a commutative operation, so `subsumes(a, b) != subsumes(b, a)`, except for the case where `a == b`.

Parameters

- **a** – a node identifier
- **b** – a node identifier

Examples

```
>>> h = MultiHierarchy('*top*', {'a': '*top*',
...                              'b': '*top*',
...                              'c': 'b'})
>>> all(h.subsumes(h.top, x) for x in h)
True
>>> h.subsumes('a', h.top)
```

(continues on next page)

(continued from previous page)

```
False
>>> h.subsumes('a', 'b')
False
>>> h.subsumes('b', 'c')
True
```

update(*subhierarchy=None*, *data=None*)

Incorporate *subhierarchy* and *data* into the hierarchy.

This method ensures that nodes are inserted in an order that does not result in an intermediate state being disconnected or cyclic, and raises an error if it cannot avoid such a state due to *subhierarchy* being invalid when inserted into the main hierarchy. Updates are atomic, so *subhierarchy* and *data* will not be partially applied if there is an error in the middle of the operation.

Parameters

- **subhierarchy** – mapping of node identifiers to parents
- **data** – mapping of node identifiers to data objects

Raises ***HierarchyError*** – when *subhierarchy* or *data* cannot be incorporated into the hierarchy

Examples

```
>>> h = MultiHierarchy('*top*')
>>> h.update({'a': '*top*'})
>>> h.update({'b': '*top*'}, data={'b': 5})
>>> h.update(data={'a': 3})
>>> h['b'] - h['a']
2
```

validate_update(*subhierarchy*, *data*)

Check if the update can apply to the current hierarchy.

This method returns (*subhierarchy*, *data*) with normalized identifiers if the update is valid, otherwise it will raise a `HierarchyError`.

Raises ***HierarchyError*** – when the update is invalid

CHAPTER

TWENTYEIGHT

DELPHIN.TOKENS

YY tokens and token lattices.

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

TWENTYNINE

DELPHIN.TSDB

Test Suite Database (TSDB) Primitives

Note: This module implements the basic, low-level functionality for working with TSDB databases. For higher-level views and uses of these databases, see *delphin.itsdb*. For complex queries of the databases, see *delphin.tsql*.

TSDB databases are plain-text file-based relational databases minimally consisting of a directory with a file, called `relations`, containing the database's schema (see *Schemas*). Every relation, or table, in the database has its own file, which may be gzipped to save space. The relations have a simple format with columns delimited by `@` and records delimited by newlines. This makes them easy to inspect at the command line with standard Unix tools such as `cut` and `awk` (but gzipped relations need to be decompressed or piped from a tool such as `zcat`).

This module handles the technical details of reading and writing TSDB databases, including:

- parsing database schemas
- transparently opening either the plain-text or gzipped relations on disk, as appropriate
- escaping and unescaping reserved characters in the data
- pairing columns with their schema descriptions
- casting types (such as `:integer`, `:date`, etc.)

Additionally, this module provides very basic abstractions of databases and relations as the *Database* and `Relation` classes, respectively. These serve as base classes for the more featureful *delphin.itsdb.TestSuite* and *delphin.itsdb.Table* classes, but may be useful as they are for simple needs.

29.1 Module Constants

`delphin.tsdb.`**`SCHEMA_FILENAME`**
`relations` – The filename for the schema.

`delphin.tsdb.`**`FIELD_DELIMITER`**
`@` – The character used to delimit fields (or columns) in a record.

`delphin.tsdb.`**`TSDB_CORE_FILES`**
The list of files used in "skeletons". Includes:

```
item
analysis
phenomenon
parameter
```

(continues on next page)

(continued from previous page)

```
set
item-phenomenon
item-set
```

delphin.tsdb.**TSDB_CODED_ATTRIBUTES**

The default values of specific fields. Includes:

```
i-wf = 1
i-difficulty = 1
polarity = -1
```

Fields without a special value given above get assigned one based on their datatype.

29.2 Schemas

A TSDB database defines its schema in a file called `relations`. This file contains descriptions of each relation (table) and its fields (columns), including the datatypes and whether a column counts as a "key". Key columns may be used when joining relations together. As an example, the first 9 lines of the `run` relation 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
  ...
```

See also:

See the TsdbSchemaRfc wiki for a description of the format of `relations` files.

In PyDelphin, TSDB schemas are represented as dictionaries of lists of `Field` objects.

class delphin.tsdb.**Field**(*name*, *datatype*, *flags=None*, *comment=None*)

A tuple describing a column in a TSDB database relation.

Parameters

- **name** (*str*) – column name
- **datatype** (*str*) – "`:string`", "`:integer`", "`:date`", or "`:float`"
- **flags** (*list*) – List of additional flags
- **comment** (*str*) – description of the column

is_key

`True` if the column is a key in the database.

Type bool

default

The default formatted value (see `format()`) when the value it describes is `None`.

Type str

`delphin.tsdb.`**`read_schema`**(*path*)

Instantiate schema dict from a schema file given by *path*.

If *path* is a directory, use the relations file under *path*. If *path* is a file, use it directly as the schema's path. Otherwise raise a `TSDBSchemaError`.

`delphin.tsdb.`**`write_schema`**(*path*, *schema*)

Serialize *schema* and write it to the relations file at *path*.

If *path* is a directory, write to a `relations` file under *path*, otherwise write to the file *path*.

`delphin.tsdb.`**`make_field_index`**(*fields*)

Create and return a mapping of field names to indices.

This mapping helps with looking up columns by their names.

Parameters **`fields`** – iterable of `Field` objects

Examples

```
>>> fields = [tsdb.Field('i-id', ':integer'),
...           tsdb.Field('i-input', ':string')]
>>> tsdb.make_field_index(fields)
{'i-id': 0, 'i-input': 1}
```

29.3 Data Operations

29.3.1 Character Escaping and Unescaping

`delphin.tsdb.`**`escape`**(*string*)

Replace any special characters with their TSDB escape sequences. The characters and their escape sequences are:

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

Also see `unescape()`

Parameters **`string`** – string to escape

Returns The escaped string

`delphin.tsdb.`**`unescape`**(*string*)

Replace TSDB escape sequences with the regular equivalents.

Also see `escape()`.

Parameters **`string`** (`str`) – TSDB-escaped string

Returns The string with escape sequences replaced

29.3.2 Record Splitting and Joining

`delphin.tsdb.`**`split`**(*line*, *fields=None*)

Split a raw line from a relation into a list of column values.

Decoding involves splitting the line by the field delimiter and unescaping special characters. The column value for empty fields is `None`.

If *fields* is given, cast each column value into its datatype, otherwise the value is returned as a string.

Parameters

- **`line`** – raw line from a TSDB relation file.
- **`fields`** – iterable of *`Field`* objects

Returns A list of column values.

`delphin.tsdb.`**`join`**`(`*values*, *fields=None*`)`

Join a list of column values into a string for a relation file.

Encoding involves escaping special characters for each value, then joining the values into a single string with the field delimiter. If *fields* is given, `None` values will be replaced with the default value for their datatype.

For creating a record from a mapping of column names to values, see *`make_record()`*.

Parameters

- **`values`** – list of column values
- **`fields`** – iterable of *`Field`* objects

Returns A TSDB-encoded string

`delphin.tsdb.`**`make_record`**`(`*colmap*, *fields*`)`

Create a record tuple from a mapping of column names to values.

This function is useful when *colmap* is either a subset or superset of the columns defined for a relation (as determined by *fields*). That is, it selects the relevant column values and fills in the missing ones with `None`. *fields* is also responsible for determining the column order.

Parameters

- **`colmap`** – mapping of column names to values
- **`fields`** – iterable of *`Field`* objects

Returns A tuple of column values

29.3.3 Datatype Conversion

`delphin.tsdb.`**`cast`**`(`*datatype*, *raw_value*`)`

Cast TSDB field *raw_value* into *datatype*.

If *raw_value* is `None` or an empty string (`''`), `None` will be returned, regardless of the *datatype*. However, when *datatype* is `:integer` and *raw_value* is `'-1'` (the default value for most `:integer` columns), `-1` is returned instead of `None`. This means that *`cast()`* is the inverse of *`format()`* except for integer values of `-1`, some date formats, and coded defaults.

Supported datatypes:

TSDB datatype	Python type
`:integer`	`int`
`:string`	`str`
`:float`	`float`
`:date`	`datetime.datetime`

Casting the `:integer`, `:string`, and `:float` types is trivial, but for `:date` TSDB uses a non-standard date format. This format generally follows the `DD-MM-YY` pattern, optionally followed by a time (with no timezone or UTF-offset allowed). The day of the month may be left unspecified, in which case `01` is used. Years may be 2 or 4 digits: in the case of 2-digit years, `19` is prepended if the 2-digit year is greater than or equal to 93 (the year of the first TSNLP publications and the earliest test suites), otherwise `20` is prepended (meaning that users are advised to start using 4-digit years by, at least, the year 2093). In addition, the more universal YYYY-MM-DD format is allowed, but it must have 4-digit years (to disambiguate with the other pattern).

Examples

```
>>> tsdb.cast(':integer', '15')
15
>>> tsdb.cast(':float', '2.05e-3')
0.00205
>>> tsdb.cast(':string', 'Abrams slept.')
'Abrams slept.'
>>> tsdb.cast(':date', '10-6-2002')
datetime.datetime(2002, 6, 10, 0, 0)
>>> tsdb.cast(':date', '8-sep-1999')
datetime.datetime(1999, 9, 8, 0, 0)
>>> tsdb.cast(':date', 'apr-95')
datetime.datetime(1995, 4, 1, 0, 0)
>>> tsdb.cast(':date', '01-dec-02 (15:31:01)')
datetime.datetime(2002, 12, 1, 15, 31, 1)
>>> tsdb.cast(':date', '2008-10-12 10:51')
datetime.datetime(2008, 10, 12, 10, 51)
```

`delphin.tsdb.`**`format`**(*datatype*, *value*, *default=None*)

Format a column *value* based on its *field*.

If *value* is `None` then *default* is returned if it is given (i.e., not `None`). If *default* is `None`, `'-1'` is returned if *datatype* is `':integer'`, otherwise an empty string (`''`) is returned.

If *datatype* is `':date'` and *value* is a `datetime.datetime` object then a TSDB-compatible date format (DD-MM-YYYY) is returned.

In all other cases, *value* is cast directly to a string and returned.

Examples

```
>>> tsdb.format(':integer', 42)
'42'
>>> tsdb.format(':integer', None)
'-1'
>>> tsdb.format(':integer', None, default='1')
'1'
>>> tsdb.format(':date', datetime.datetime(1999,9,8))
'8-sep-1999'
```

29.4 File and Directory Operations

29.4.1 Paths

delphin.tsdb.**is_database_directory**(*path*)

Return `True` if *path* is a valid TSDB database directory.

A path is a valid database directory if it is a directory containing a schema file. This is a simple test; the schema file itself is not checked for validity.

delphin.tsdb.**get_path**(*dir*, *name*)

Determine if the file path should end in .gz or not and return it.

A .gz path is preferred only if it exists and is newer than any regular text file path.

Parameters

- **dir** – TSDB database directory
- **name** – name of a file in the database

Raises *TSDBError* – when neither the .gz nor the text file exist.

29.4.2 Relation File Access

delphin.tsdb.**open**(*dir*, *name*, *encoding=None*)

Open a TSDB database file.

Unlike a normal `open()` call, this function takes a base directory *dir* and a filename *name* and determines whether the plain text *dir/name* or compressed *dir/name*.gz file is opened. Furthermore, this function only opens files in read-only text mode. For writing database files, see *write()*.

Parameters

- **dir** – path to the database directory
- **name** – name of the file to open
- **encoding** – character encoding of the file

Example

```
>>> sentences = []
>>> with tsdb.open('my-profile', 'item') as item:
...     for line in item:
...         sentences.append(tsdb.split(line)[6])
```

delphin.tsdb.**write**(*dir*, *name*, *records*, *fields=None*, *append=False*, *gzip=False*, *encoding='utf-8'*)

Write *records* to relation *name* in the database at *dir*.

The simplest way to write data to a file would be something like the following:

```
>>> with open(os.path.join(db.path, 'item'), 'w') as fh:
...     print('\n'.join(map(tsdb.join, db['item'])), file=fh)
```

This function improves on that method by doing the following:

- Determining the path from the *gzip* parameter and existing files

- Writing plain text or compressed data, as appropriate
- Appending or overwriting data, as requested
- Using the schema information to format fields
- Writing to a temporary file then copying when done; this prevents accidental data loss when overwriting a file that is being read
- Deleting any alternative (compressed or plain text) file to avoid having inconsistent files (e.g., delete any existing `item` when writing `item.gz`)

Note that *append* cannot be used with *gzip* or with an existing gzipped file and in such a case a `NotImplementedError` will be raised. This may be allowed in the future, but as appending to a gzipped file (in general) results in inefficient compression, it is better to append to plain text and compress when done.

Parameters

- **dir** – path to the database directory
- **name** – name of the relation to write
- **records** – iterable of records to write
- **fields** – iterable of *Field* objects, optional if *dir* points to an existing test suite directory
- **append** – if `True`, append to rather than overwrite the file
- **gzip** – if `True` and the file is not empty, compress the file with `gzip`; if `False`, do not compress
- **encoding** – character encoding of the file

Example

```
>>> tsdb.write('my-profile',
...            'item',
...            item_records,
...            schema['item'])
```

29.4.3 Database Directories

delphin.tsdb.**initialize_database**(*path*, *schema*, *files=False*)

Initialize a bare database directory at *path*.

Initialization creates the directory at *path* if it does not exist, writes the schema, an deletes any existing files defined by the schema.

Warning: If *path* points to an existing directory, all relation files defined by the schema will be overwritten or deleted.

Parameters

- **path** – the path to the destination database directory
- **schema** – the destination database schema
- **files** – if `True`, create an empty file for every relation in *schema*

`delphin.tsdb.`**`write_database`**(*db, path, names=None, schema=None, gzip=False, encoding='utf-8'*)

Write TSDB database *db* to *path*.

If *path* is an existing file (not a directory), a `TSDBError` is raised. If *path* is an existing directory, the files for all relations in the destination schema will be cleared. Every relation name in *names* must exist in the destination schema. If *schema* is given (even if it is the same as for *db*), every record will be remade (using `make_record()`) using the schema, and columns may be dropped or `None` values inserted as necessary, but no more sophisticated changes will be made.

> **Warning:** If *path* points to an existing directory, all relation files defined by the schema will be overwritten or deleted.

Parameters

- **db** – Database containing data to write
- **path** – the path to the destination database directory
- **names** – list of names of relations to write; if `None` use all relations in the destination schema
- **schema** – the destination database schema; if `None` use the schema of *db*
- **gzip** – if `True`, compress all non-empty files; if `False`, do not compress
- **encoding** – character encoding for the database files

29.5 Basic Database Class

class `delphin.tsdb.`**`Database`**(*path, autocast=False, encoding='utf-8'*)

A basic abstraction of a TSDB database.

This class manages basic access into a TSDB database by loading its schema and allowing for named access to relation data.

> **Warning:** Named access to relation data returns a generator iterator of an open file. Calling `generator.close()` or using an idiom like `contextlib.closing()` ensures that the file descriptor gets closed.

Parameters

- **path** – path to the database directory
- **autocast** – if `True`, automatically cast column values to their datatypes
- **encoding** – character encoding of the database files

Example

```
>>> db = tsdb.Database('my-profile')
>>> items = db['item']
>>> first_record = next(items)
>>> items.close()
```

schema
The schema for the database.

autocast
Whether to automatically cast column values to their datatypes.

encoding
The character encoding of database files.

property path
The database directory's path.

select_from(*name*, *columns=None*, *cast=False*)
Yield values for *columns* from relation *name*.

29.6 Exceptions

exception delphin.tsdb.**TSDBSchemaError**(**args*, ***kwargs*)
Bases: *delphin.tsdb.TSDBError*

Raised when there is an error processing a TSDB schema.

exception delphin.tsdb.**TSDBError**(**args*, ***kwargs*)
Bases: *delphin.exceptions.PyDelphinException*

Raised when encountering invalid TSDB databases.

exception delphin.tsdb.**TSDBWarning**(**args*, ***kwargs*)
Bases: *delphin.exceptions.PyDelphinWarning*

Raised when encountering possibly invalid TSDB data.

CHAPTER

THIRTY

DELPHIN.TSQL

See also:

The *select* command is a quick way to query test suites with TSQL queries.

TSQL – Test Suite Query Language

Note: This module deals with queries of TSDB databases. For basic, low-level access to the databases, see `delphin.tsdb`. For high-level operations and structures on top of the databases, see `delphin.itsdb`.

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 <relations>] [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 relations.

The optional `from` clause provides a list of relation 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 relations (e.g., `i-id from output`). Alternatively, qualified names (e.g., `item.i-id`) can specify both the column and the relation at the same time.

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` relation
- `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:

- qualified column names (e.g., `item.i-id`)
- multiple `where` clauses (as described above)

30.1 Module Functions

`delphin.tsql.`**`inspect_query`**(*querystring*)

Parse *querystring* and return the interpreted query dictionary.

Example

```
>>> from delphin import tsql
>>> from pprint import pprint
>>> pprint(tsql.inspect_query(
...     'select i-input from item where i-id < 100'))
{'type': 'select',
 'projection': ['i-input'],
 'relations': ['item'],
 'condition': ('<', ('i-id', 100))}
```

`delphin.tsql.`**`query`**(*querystring*, *db*, ***kwargs*)

Perform query *querystring* on the testsuite *ts*.

Note: currently only 'select' queries are supported.

Parameters

- **querystring** (*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**(*querystring*, *db*, *record_class=None*)

Perform the TSQL selection query *querystring* on testsuite *ts*.

Note: The `select`/`retrieve` part of the query is not included.

Parameters

- **querystring** – TSQL select query
- **db** – TSDB database to query over

Example

```
>>> list(tsql.select('i-id where i-length < 4', ts))
[[142], [1061]]
```

30.2 Exceptions

exception delphin.tsql.**TSQLSyntaxError**(*message=None*, *filename=None*, *lineno=None*, *offset=None*, *text=None*)

Bases: *delphin.exceptions.PyDelphinSyntaxError*

Raised when encountering an invalid TSQL query.

exception delphin.tsql.**TSQLError**(**args*, ***kwargs*)

Bases: *delphin.exceptions.PyDelphinException*

Raised on invalid TSQL operations.

CHAPTER

THIRTYONE

DELPHIN.VARIABLE

Functions for working with MRS variables.

This module contains functions to inspect the type and identifier of variables (`split()`, `type()`, `id()`) and check if a variable string is well-formed (`is_valid()`). It additionally has constants for the standard variable types: `UNSPECIFIC`, `INDIVIDUAL`, `INSTANCE_OR_HANDLE`, `EVENTUALITY`, `INSTANCE`, and `HANDLE`. Finally, the `VariableFactory` class may be useful for tasks like DMRS to MRS conversion for managing the creation of new variables.

31.1 Variables in MRS

Variables are a concept in Minimal Recursion Semantics coming from formal semantics. Consider this logical form for a sentence like "the dog barks":

```
x(dog(x) ^ bark(x))
```

Here *x* is a variable that represents an entity that has the properties that it is a dog and it is barking. Davidsonian semantics introduce variables for events as well:

```
ex(dog(x) ^ bark(e, x))
```

MRS uses variables in a similar way to Davidsonian semantics, except that events are not explicitly quantified. That might look like the following (if we ignore quantifier scope underspecification):

```
the(x4) [dog(x4)] {bark(e2, x4)}
```

"Variables" are also used for scope handles and labels, as in this minor modification that indicates the scope handles:

```
h3:the(x4) [h6:dog(x4)] {h1:bark(e2, x4)}
```

There is some confusion of terminology here. Sometimes "variable" is contrasted with "handle" to mean an instance (`x`) or eventuality (`e`) variable, but in this module "variable" means the identifiers used for instances, eventualities, handles, and their supertypes.

The form of MRS variables is the concatenation of a variable *type* (also called a *sort*) with a variable *id*. For example, the variable type `e` and id `2` form the variable `e2`. Generally in MRS the variable ids, regardless of the type, are unique, so for instance one would not see `x2` and `e2` in the same structure.

The variable types are arranged in a hierarchy. While the most accurate variable type hierarchy for a particular grammar is obtained via its SEM-I (see `delphin.semi`), in practice the standard hierarchy given below is used by all DELPH-IN grammars. The hierarchy in TDL would look like this (with an ASCII rendering in comments on the right):

```
u := *top*.  ;      u
i := u.      ;     / \
p := u.      ;    i   p
e := i.      ;   / \ / \
x := i & p.  ; e    x    h
h := p.
```

In PyDelphin the equivalent hierarchy could be created as follows with a *delphin.hierarchy.MultiHierarchy*:

```
>>> from delphin import hierarchy
>>> h = hierarchy.MultiHierarchy(
...     '*top*',
...     {'u': '*top*',
...      'i': 'u',
...      'p': 'u',
...      'e': 'i',
...      'x': 'i p',
...      'h': 'p'}
... )
```

31.2 Module Constants

`delphin.variable.`**`UNSPECIFIC`**
: `u` – The unspecific (or unbound) top-level variable type.

`delphin.variable.`**`INDIVIDUAL`**
: `i` – The variable type that generalizes over eventualities and instances.

`delphin.variable.`**`INSTANCE_OR_HANDLE`**
: `p` – The variable type that generalizes over instances and handles.

`delphin.variable.`**`EVENTUALITY`**
: `e` – The variable type for events and other eventualities (adjectives, adverbs, prepositions, etc.).

`delphin.variable.`**`INSTANCE`**
: `x` – The variable type for instances and nominal things.

`delphin.variable.`**`HANDLE`**
: `h` – The variable type for scope handles and labels.

31.3 Module Functions

`delphin.variable.`**`split`**(*var*)
: Split a valid variable string into its variable type and id.

Examples

```
>>> variable.split('h3')
('h', '3')
>>> variable.split('ref-ind12')
('ref-ind', '12')
```

`delphin.variable.`**`type`**`(`*var*`)`

Return the type (i.e., sort) of a valid variable string.

sort() is an alias for *type()*.

Examples

```
>>> variable.type('h3')
'h'
>>> variable.type('ref-ind12')
'ref-ind'
```

`delphin.variable.`**`sort`**`(`*var*`)`

sort() is an alias for *type()*.

`delphin.variable.`**`id`**`(`*var*`)`

Return the integer id of a valid variable string.

Examples

```
>>> variable.id('h3')
3
>>> variable.id('ref-ind12')
12
```

`delphin.variable.`**`is_valid`**`(`*var*`)`

Return `True` if *var* is a valid variable string.

Examples

```
>>> variable.is_valid('h3')
True
>>> variable.is_valid('ref-ind12')
True
>>> variable.is_valid('x')
False
```

31.4 Classes

`class` `delphin.variable.`**`VariableFactory`**`(`*starting_vid=1*`)`

Simple class to produce variables by incrementing the variable id.

This class is intended to be used when creating an MRS from a variable-less representation like DMRS where the variable types are known but no variable id is assigned.

Parameters **`starting_vid`** (*int*) – the id of the first variable

`vid`

the id of the next variable produced by *new()*

Type int

`index`

a mapping of ids to variables

Type dict

store

a mapping of variables to associated properties

Type dict

new(*type*, *properties=None*)

Create a new variable for the given *type*.

Parameters

- **type** (*str*) – the type of the variable to produce
- **properties** (*list*) – properties to associate with the variable

Returns A (variable, properties) tuple

CHAPTER

THIRTYTWO

DELPHIN.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

32.1 Module functions

`delphin.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

32.2 Classes

`class` `delphin.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

32.3 Exceptions

exception delphin.vpm.**VPMSyntaxError**(*message=None*, *filename=None*, *lineno=None*, *offset=None*, *text=None*)

Bases: *delphin.exceptions.PyDelphinSyntaxError*

Raised when loading an invalid VPM.

CHAPTER

THIRTYTHREE

DELPHIN.WEB

Client interfaces and a server for the DELPH-IN Web API.

33.1 delphin.web.client

DELPH-IN Web API Client

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). This dependency is satisfied if you install PyDelphin with the `[web]` extra (see *Requirements, Installation, and Testing*).

Basic access is available via the `parse()`, `parse_from_iterable()`, `generate()`, and `generate_from_iterable()` functions:

```
>>> from delphin.web import client
>>> url = 'http://erg.delph-in.net/rest/0.9/'
>>> client.parse('Abrams slept.', server=url)
Response({'input': 'Abrams slept.', 'readings': 1, 'results': [{'result-id': 0}],
↪'tcpu': 7, 'pedges': 17})
>>> client.parse_from_iterable(['Abrams slept.', 'It rained.'], server=url)
<generator object parse_from_iterable at 0x7f546661c258>
>>> client.generate('[ LTOP: 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 CARG: "Abrams" ARG0: x3 ]  [ _sleep_v_
↪1<7:13> LBL: h1 ARG0: e2 ARG1: x3 ] > HCONS: < h0 qeq h1 h5 qeq h7 > ICONS: < > ]')
Response({'input': '[ LTOP: 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 CARG: "Abrams" ARG0: x3 ]  [ _sleep_v_
↪1<7:13> LBL: h1 ARG0: e2 ARG1: x3 ] > HCONS: < h0 qeq h1 h5 qeq h7 > ICONS: < > ]',
↪'readings': 1, 'results': [{'result-id': 0, 'surface': 'Abrams slept.'}], 'tcpu': 8,
↪ 'pedges': 59})
```

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 instantiate and use subclasses of `Client`, which manages the connections to a server. They can also be used directly:

```
>>> parser = web.Parser(server=url)
>>> parser.interact('Dogs chase cats.')
Response({'input': 'Dogs chase cats.', ...
>>> generator = web.Generator(server=url)
>>> generator.interact('[ LTOP: h0 INDEX: e2 ...')
Response({'input': '[ LTOP: h0 INDEX: e2 ...', ...)
```

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

```
>>> r = parser.interact('Dogs chase cats', params={'mrs':'json'})
>>> r.result(0)
Result({'result-id': 0, 'score': 0.5938, ...
>>> r.result(0)['mrs']
{'variables': {'h1': {'type': 'h'}, 'x6': ...
>>> r.result(0).mrs()
<MRS object (udef_q dog_n_1 chase_v_1 udef_q cat_n_1) at 140000394933248>
```

If PyDelphin does not support deserialization for a format provided by the server (e.g. LaTeX output), the `Result` object raises a `TypeError`.

33.1.1 Client Functions

`delphin.web.client.`**`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 (LOGON's ERG server is used by default)
- **`params`** (*`dict`*) – a dictionary of request parameters
- **`headers`** (*`dict`*) – a dictionary of additional request headers

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

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

`delphin.web.client.`**`parse_from_iterable`**`(`*inputs*, *server='http://erg.delph-in.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 (LOGON's ERG server is used by default)
- **`params`** (*`dict`*) – a dictionary of request parameters
- **`headers`** (*`dict`*) – a dictionary of additional request headers

Yields Response objects for each successful response.

Raises **`requests.HTTPError`** – for the first response with a status code that is not 200 `delphin.web.client.`**`generate`**(*input*, *server='http://erg.delph-in.net/rest/0.9/'*, *params=None*, *headers=None*)

Request realizations for *input*.

Parameters

- **input** (*str*) – SimpleMRS to be realized
- **server** (*str*) – the url for the server (LOGON's ERG server is used by default)
- **params** (*dict*) – a dictionary of request parameters
- **headers** (*dict*) – a dictionary of additional request headers

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

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

`delphin.web.client.`**`generate_from_iterable`**(*inputs*, *server='http://erg.delph-in.net/rest/0.9/'*, *params=None*, *headers=None*)

Request realizations for all *inputs*.

Parameters

- **inputs** (*iterable*) – SimpleMRS strings to realize
- **server** (*str*) – the url for the server (LOGON's ERG server is used by default)
- **params** (*dict*) – a dictionary of request parameters
- **headers** (*dict*) – a dictionary of additional request headers

Yields Response objects for each successful response.

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

33.1.2 Client Classes

class `delphin.web.client.`**`Client`**(*server*)

Bases: `delphin.interface.Processor`

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

Note: This class is not meant to be used directly. Use a subclass instead.

`interact`(*datum*, *params=None*, *headers=None*)

Request the server to process *datum* return the response.

Parameters

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

Returns A Response 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 server and return the response with context.

The *keys* parameter can be used to track item identifiers through a Web API interaction. If the `task` member is set on the Client instance (or one of its subclasses), it is kept in the response as well.

Parameters

- **datum** (*str*) – the input sentence or MRS
- **keys** (*dict*) – a mapping of item identifier names and values
- **params** (*dict*) – a dictionary of request parameters
- **headers** (*dict*) – a dictionary of additional request headers

Returns *Response*

class delphin.web.client.**Parser**(*server*)
Bases: *delphin.web.client.Client*

A class for managing parse requests to a Web API server.

class delphin.web.client.**Generator**(*server*)
Bases: *delphin.web.client.Client*

A class for managing generate requests to a Web API server.

33.2 delphin.web.server

DELPH-IN Web API Server

This module provides classes and functions that implement a subset of the DELPH-IN Web API DELPH-IN Web API described here:

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

Note: Requires Falcon (https://falcon.readthedocs.io/). This dependency is satisfied if you install PyDelphin with the `[web]` extra (see *Requirements, Installation, and Testing*).

In addition to the parsing API, this module also provides support for generation and for browsing [incr tsdb()] test suites. In order to use it, you will need a WSGI server such as gunicorn, mod_wsgi for Apache2, etc. You then write a WSGI stub for the server to use, such as the following example:

```
# file: wsgi.py

import falcon

from delphin.web import server

application = falcon.API()

server.configure(
    application,
    parser='~/grammars/erg-2018-x86-64-0.9.30.dat',
    generator='~/grammars/erg-2018-x86-64-0.9.30.dat',
    testsuites={
        'gold': [
            {'name': 'mrs', 'path': '~/grammars/erg/tsdb/gold/mrs'}
        ]
```

(continues on next page)

(continued from previous page)

```
    }
)
```

You can then run a local instance using, for instance, gunicorn:

```
$ gunicorn wsgi
[2019-07-12 16:03:28 +0800] [29920] [INFO] Starting gunicorn 19.9.0
[2019-07-12 16:03:28 +0800] [29920] [INFO] Listening at: http://127.0.0.1:8000 (29920)
[2019-07-12 16:03:28 +0800] [29920] [INFO] Using worker: sync
[2019-07-12 16:03:28 +0800] [29923] [INFO] Booting worker with pid: 29923
```

And make requests with, for instance, **curl**:

```
$ curl 'http://127.0.0.1:8000/parse?input=Abrams%20slept.&mrs' -v
*   Trying 127.0.0.1...
* TCP_NODELAY set
* Connected to 127.0.0.1 (127.0.0.1) port 8000 (#0)
> GET /parse?input=Abrams%20slept.&mrs HTTP/1.1
> Host: 127.0.0.1:8000
> User-Agent: curl/7.61.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Server: gunicorn/19.9.0
< Date: Fri, 12 Jul 2019 08:04:29 GMT
< Connection: close
< content-type: application/json
< content-length: 954
<
* Closing connection 0
{"input": "Abrams slept.", "readings": 1, "results": [{"result-id": 0, "mrs": {"top":
↪"h0", "index": "e2", "relations": [{"label": "h4", "predicate": "proper_q",
↪"arguments": {"ARG0": "x3", "RSTR": "h5", "BODY": "h6"}, "lnk": {"from": 0, "to": 6}
↪}, {"label": "h7", "predicate": "named", "arguments": {"CARG": "Abrams", "ARG0": "x3
↪"}, "lnk": {"from": 0, "to": 6}}, {"label": "h1", "predicate": "_sleep_v_1",
↪"arguments": {"ARG0": "e2", "ARG1": "x3"}, "lnk": {"from": 7, "to": 13}}],
↪"constraints": [{"relation": "qeq", "high": "h0", "low": "h1"}, {"relation": "qeq",
↪"high": "h5", "low": "h7"}], "variables": {"e2": {"type": "e", "properties": {"SF":
↪"prop", "TENSE": "past", "MOOD": "indicative", "PROG": "-", "PERF": "-"}}, "x3": {
↪"type": "x", "properties": {"PERS": "3", "NUM": "sg", "IND": "+"}}, "h5": {"type":
↪"h"}, "h6": {"type": "h"}, "h0": {"type": "h"}, "h1": {"type": "h"}, "h7": {"type":
↪"h"}, "h4": {"type": "h"}}}}], "tcpu": 7, "pedges": 17}
```

33.2.1 Module Functions

`delphin.web.server.`**`configure`**(*api*, *parser=None*, *generator=None*, *testsuites=None*)

Configure server application *api*.

This is the preferred way to setup the server application, but the task-specific classes defined in this module can also be used to setup custom routes, for instance.

If a path is given for *parser* or *generator*, it will be used to construct a `ParseServer` or `GenerationServer` instance, respectively, with default arguments to the underlying `ACEProcessor`. If non-default arguments are needed, pass in the customized `ParseServer` or `GenerationServer` instances directly.

Parameters

- **api** – an instance of `falcon.API`
- **parser** – a path to a grammar or a *ParseServer* instance
- **generator** – a path to a grammar or a *GenerationServer* instance
- **testsuites** – mapping of collection names to lists of test suite entries

Example

```
>>> server.configure(
...     api,
...     parser='~/grammars/erg-2018-x86-64-0.9.30.dat',
...     testsuites={
...         'gold': [
...             {'name': 'mrs',
...              'path': '~/grammars/erg/tsdb/gold/mrs'}]})
```

33.2.2 Server Application Classes

class delphin.web.server.**ProcessorServer**(*grammar*, **args*, ***kwargs*)
A server for results from an ACE processor.

Note: This class is not meant to be used directly. Use a subclass instead.

class delphin.web.server.**ParseServer**(*grammar*, **args*, ***kwargs*)
Bases: *delphin.web.server.ProcessorServer*

A server for parse results from ACE.

processor_class
alias of *delphin.ace.ACEParser*

class delphin.web.server.**GenerationServer**(*grammar*, **args*, ***kwargs*)
Bases: *delphin.web.server.ProcessorServer*

A server for generation results from ACE.

processor_class
alias of *delphin.ace.ACEGenerator*

class delphin.web.server.**TestSuiteServer**(*testsuites*, *transforms=None*)
A server for a collection of test suites.

Parameters

- **testsuites** – list of test suite descriptions
- **transforms** – mapping of table names to lists of (column, transform) pairs.

CHAPTER

THIRTYFOUR

API REFERENCE:

34.1 Core API

- *delphin.exceptions*
- *delphin.hierarchy* – Multiple-inheritance hierarchies
- *delphin.codecs* – Serialization codecs
- *delphin.commands*

34.2 Interfacing External Tools

- *delphin.interface*
- *delphin.ace* – ACE
- *delphin.web* – DELPH-IN Web API

34.3 Tokenization

- *delphin.lnk* – Surface alignment
- *delphin.repp* – Regular Expression Preprocessor
- *delphin.tokens* – YY token lattices

34.4 Syntax

- *delphin.derivation* – UDF/UDX derivation trees

34.5 Semantics

- *delphin.dmrs* – Dependency Minimal Recursion Semantics
- *delphin.eds* – Elementary Dependency Structures
- *delphin.predicate* – Semantic predicates

- *delphin.mrs* – Minimal Recursion Semantics
- *delphin.sembase*
- *delphin.semi* – Semantic Interface (or model)
- *delphin.scope* – Scope operations
- *delphin.variable*
- *delphin.vpm* – Variable property mapping

34.6 Test Suites

- *delphin.itsdb* – [incr tsdb()]
- *delphin.tsdb* – Test Suite Database
- *delphin.tsql* – Test Suite Query Language

34.7 Grammars

- *delphin.tdl* – Type Description Language
- *delphin.tfs* – Typed feature structures

CHAPTER

THIRTYFIVE

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[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.

[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.

[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.

[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

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y