Penman Documentation

Release v0.7.2

Michael Wayne Goodman

Dec 12, 2019

CONTENTS:

The Penman package is a library for working with graphs in the PENMAN format. Its primary job is thus parsing the serialized form into an internal `graph` representation and format graphs into the serialized form again. Once parsed, the graphs can be inspected and manipulated, depending on one's needs.

The interpretation of PENMAN into the internal graph depends on a semantic model. The default `model` works in most cases, but for people working with Abstract Meaning Representation (AMR) data, the `AMR model` will allow them to perform operations in a way that follows the principles of AMR. Users may also define custom models if they need more control.

CHAPTER

ONE

INSTALLATION AND SETUP

Penman releases are available on PyPI and the source code is on GitHub. Users of Penman will probably want to install from PyPI using **pip** as it is the easiest method and it makes the **penman** command available at the command line. Developers and contributors of Penman will probably want to install from the source code.

1.1 Requirements

The Penman package runs with Python 3.6 and higher versions, but otherwise it has no dependencies beyond Python's standard library.

Some development tasks, such as unit testing, building the documentation, or making releases, have additional dependencies. See *Installing from Source* for more information.

1.2 Installation

1.2.1 Installing from PyPI

Install the latest version from PyPI using **pip** (using a virtual environment is recommended):

```
$ pip install penman
```

After running the above command, the `penman` module will be available in Python and the **penman** command will be available at the command line.

1.2.2 Installing from Source

Developers and contributors of the Penman project may wish to install from the source code using one of several "extras", which are given in brackets after the package name. The available extras are:

- `test` – install dependencies for unit testing
- `doc` – install dependencies for building the documentation
- `dev` – install dependencies for both of the above plus those needed for publishing releases

When installing from source code, the `-e` option is also useful as any changes made to the source code after the install will be reflected at runtime (otherwise one needs to reinstall after any changes). The following is how one might clone the source code, create and activate a virtual environment, and install for development:

```
$ git clone https://github.com/goodmami/penman.git
[...]
$ cd penman/
$ python3 -m venv env
$ source env/bin/activate
(env) $ pip install -e .[dev]
```

1.3 Testing

1.3.1 Unit Testing with pytest

The unit tests can be run with pytest from the project directory of the source code:

```
(env) $ pytest
```

For testing multiple Python versions, a tool like tox can automate the creation and activation of multiple virtual environments.

1.3.2 Type-checking with Mypy

The Penman project heavily uses **PEP 484** and **PEP 526** type annotations for static type checking. The code can be type-checked using Mypy:

```
(env) $ mypy penman
```

1.3.3 Style-checking with Flake8

Flake8 is used for style checking with the following checks disabled:

- E241 – large data descriptions are easier to read with whitespace
- W503 – binary operators should appear after a line break

```
(env) $ flake8 --ignore=E241,W503 penman
```

CHAPTER TWO

BASIC USAGE

This document will give an overview of how to use Penman as a tool and as a library. For motivation, here's an example of its tool usage:

```
$ penman --indent 3 --compact <<< '(s / sleep :polarity - :ARG0 (i / i))'
(s / sleep :polarity -
   :ARG0 (i / i))
```

And here's an example of its library usage:

```
>>> from penman import PENMANCodec
>>> codec = PENMANCodec()
>>> g = codec.decode('(s / sleep-01 :polarity - :ARG0 (i / i))')
>>> g.triples.remove(('s', ':polarity', '-'))
>>> print(PENMANCodec().encode(g))
(s / sleep-01
   :ARG0 (i / i))
```

2.1 Using Penman as a Tool

Once installed (see *Installation and Setup*), the **penman** command becomes available. It allows you to perform some basic tasks with PENMAN graphs without having to write any Python code. Run **penman --help** to get a synposis of its usage:

```
$ penman --help
usage: penman [-h] [-V] [-v] [-q] [--model FILE | --amr] [--indent N]
              [--compact] [--triples] [--rearrange KEY] [--canonicalize-roles]
              [--reify-edges] [--reify-attributes] [--indicate-branches]
              [FILE [FILE ...]]

Read and write graphs in the PENMAN notation.

positional arguments:
  FILE                  read graphs from FILEs instead of stdin

optional arguments:
  -h, --help            show this help message and exit
  -V, --version         show program's version number and exit
  -v, --verbose         increase verbosity
  -q, --quiet           suppress output on <stdout> and <stderr>
  --model FILE          JSON model file describing the semantic model
  --amr                 use the AMR model
```

(continues on next page)

(continued from previous page)

```
formatting options:
  --indent N            indent N spaces per level ("no" for no newlines)
  --compact             compactly print node attributes on one line
  --triples             print graphs as triple conjunctions

normalization options:
  --rearrange KEY       sort or randomize the order of relations on each node
  --canonicalize-roles  canonicalize role forms
  --reify-edges         reify all eligible edges
  --reify-attributes    reify all attributes
  --indicate-branches   insert triples to indicate tree structure
```

The **penman** command can read input from stdin or from one or more files. Currently it always outputs to stdout. Options are available to customize the formatting of the output, such as for controlling indentation. Normalization options allow one to transform the graph in predefined ways prior to serialization. For example:

```
$ penman --amr --indent=3 --reify-edges <<< '(a / apple :quant 3)'
(a / apple
   :ARG1-of (_ / have-quant-91
      :ARG2 3))
```

2.2 Using Penman as a Library

While the command-line utility is convenient, it does not expose all the functionality that the Penman package has. For more sophisticated uses, the API allows one to directly inspect trees and graphs, construct and manipulate trees and graphs, further customize serialization, interface with other systems, etc.

For example:

```
>>> from penman import PENMANCodec
>>> codec = PENMANCodec()
>>> g = codec.decode('(b / bark-01 :ARG0 (d / dog))')
>>> g.attributes()
[Attribute(source='b', role=':instance', target='bark-01'), Attribute(source='d',␣
↪role=':instance', target='dog')]
>>> g.edges()
[Edge(source='b', role=':ARG0', target='d')]
>>> g.variables()
{'d', 'b'}
>>> print(codec.encode(g, top='d'))
(d / dog
   :ARG0-of (b / bark-01))
>>> g.triples.append(('b', ':polarity', '-'))
>>> print(codec.encode(g))
(b / bark-01
   :ARG0 (d / dog)
   :polarity -)
```

Importing directly from the *penman* module allows for basic usage of the library, but anything more advanced can take advantage of the full API. See the *API documentation* for more information.

CHAPTER

THREE

PENMAN NOTATION

PENMAN notation, originally called *Sentence Plan Notation* in the PENMAN project ([KAS1989]), is a serialization format for the directed, rooted graphs used to encode semantic dependencies, most notably in the Abstract Meaning Representation (AMR) framework. It looks similar to Lisp's S-Expressions in using parentheses to indicate nested structures. For example, here is an AMR for "He drives carelessly.":

```
(d / drive-01
   :ARG0 (h / he)
   :manner (c / care-04
              :polarity -))
```

Let's break that down a bit:

```
;    ─────────────────────────── Variable (this one is the graph's top)
;    │ ───────────────────────── Indicates the node's concept
;    │ │    ──────────────────── Concept (node label)
;      ────────
    (d / drive-01
;            ─────────────────── Edge relation
;      ─────────────
       :ARG0 (h / he)
;      ────
;        └─────────────────────── Role (edge label)
       :manner (c / care-04
;                       ──────── Attribute relation
;                  ──────────
                  :polarity -))
;
;                           └──── Atom (or "constant")
```

The linearized form can only describe projective structures such as trees, so in order to capture non-projective graphs, nodes get identifiers (called *variables*; e.g., d, h, and c above) which can be referred to later to establish a reentrancy.

PENMAN notation can be very roughly described with the following BNF grammar (from [GOO2019]):

```
<node> ::= '(' <id> '/' <node-label> <edge>* ')'
<edge> ::= ':'<edge-label> (<const>|<id>|<node>)
```

A more complete description is given by the following PEG grammar. In addition to being more complete, it also extends the grammar to allow for surface alignments.

```
# Syntactic productions (whitespace is allowed around non-terminals)
Start     <- Node
Node      <- '(' Variable NodeLabel? Relation* ')'
NodeLabel <- '/' Concept Alignment?
```

(continues on next page)

(continued from previous page)

```
Concept   <- Atom
Relation  <- Role Alignment? (Node / Atom Alignment?)
Atom      <- Variable / Constant
Constant  <- String / Float / Integer / Symbol
Variable  <- Symbol

# Lexical productions (whitespace is not allowed)
Symbol    <- NameChar+
Role      <- ':' NameChar*
Alignment <- '~' ([a-zA-Z] '.'?)? Digit+ (',' Digit+)*
String    <- '"' (!'"' ('\\' . / .))* '"'
Float     <- Decimal Exponent? / Integer Exponent
Decimal   <- [-+]? (Digit+ '.' Digit* / '.' Digit+ )
Exponent  <- [eE] Integer
Integer   <- [-+]? Digit+
NameChar  <- ![ \n\t\r\f\v()/,:~] .
Digit     <- [0-9]
```

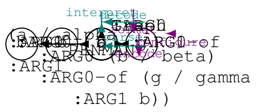
This grammar has some seemingly unnecessary ambiguity in that both the `Variable` and `Constant` alternatives for `Atom` can resolve to `Symbol`, but it is written this way to accommodate syntax variants that further restrict the form of variables. Also, the distinction between edge relations and attribute relations is semantic: if the target of a relation is the variable of some other node, then it is an edge, otherwise it is an attribute.

CHAPTER

FOUR

TREES, GRAPHS, AND EPIGRAPHS

On the surface, the structures encoded in PENMAN Notation (see *here*) are a tree, and only by resolving repeated node identifiers (variables) as reentrancies does the actual graph become accessible. The Penman library thus accommodates the three stages of a structure: the linear PENMAN string, the surface `tree`, and the pure `graph`. Going from a string to a tree is called **parsing**, and from a tree to a graph is **interpretation**, while the whole process (string to graph) is called **decoding**. Going from a graph to a tree is called **configuration**, and from a tree to a string is **formatting**, while the whole process is called **encoding**. These processes are illustrated by the following figure:



Conversion from a PENMAN string to a `Tree`, and vice versa, is straightforward and lossless. Conversion to a `Graph`, however, is potentially lossy as the same graph can be represented by different trees. For example, the graph in the figure above could be serialized to any of these PENMAN strings:

```
(a / alpha                 (a / alpha                     (a / alpha
   :ARG0 (b / beta)           :ARG0 (b / beta                :ARG0 (b / beta
   :ARG0-of (g / gamma           :ARG1-of (g / gamma))          :ARG1-of (g / gamma
      :ARG1 b))               :ARG0-of g)                          :ARG0 a)))
```

Even more serializations are possible if you do not require the first occurrence of a variable to define the node (with its node label (concept) and outgoing edges), or if you allow other nodes to be the top.

The Penman library therefore introduces the concept of the **epigraph** (not to be confused with other senses of *epigraph*, such as an inscription on a building or a passage at the beginning of a book), which is information on top of the graph that instructs the `codec` how the graph should be serialized. The epigraph is thus analagous to the idea of the epigenome: epigenetic markers controls how genes are expressed in an individual as the epigraphical markers control how graph triples are expressed in a tree or string. Separating the graph and the epigraph thus allow the graph to be a pure representation of the triples expressed in a PENMAN serialization without losing information about the surface form.

There are currently two kinds of epigraphical markers: layout markers and surface alignment markers. Surface alignment markers are parsed from the string and stored in the tree then propagated to the graph upon interpretation. Layout markers are created when the tree is interpreted into a graph. When an edge goes to a new node and not a constant or variable, a `Push` marker is inserted. When a node ends, a `POP` marker is inserted. With these markers, and the ordering of triples, the graph can be configured to a specific tree structure.

CHAPTER

FIVE

NOTES ON SERIALIZATION

A PENMAN-serialized graph takes the form of a tree with labeled reentrancies, so in deserialization it is first parsed directly into a tree and then the pure graph is interpreted from it.

```
(b / bark
   :ARG0 (d / dog))
```

The above PENMAN string is parsed to the following tree:

```
Tree(('b', [(':instance', 'bark', []),
            (':ARG0', ('d', [(':instance', 'dog', [])]), [])]))
```

The structure of a tree node is `(var, branches)` while the structure of a branch is `(role, target, epidata)`. The target of a branch can be an atomic value or a tree node. The epidata field is a list of epigraphical markers. This tree is then interpreted to the following triples:

```
Graph(triples=[
       ('b', ':instance', 'bark'),
       ('b', ':ARG0', 'd'),
       ('d', ':instance', 'dog')
      ],
      epidata={
       ('b', ':ARG0', 'd'): [Push('d')],
       ('d', ':instance', 'dog'): [POP]
      })
```

Serialization goes in the reverse order: from a pure graph to a tree to a string.

5.1 Allowed Graphs

The Penman library robustly allows some kinds of invalid and unconventional graphs.

Unproblematic:

```
# Normal
(a / a-label :ROLE (b / b-label))

# Unlabeled nodes, edges
(a :ROLE (b))
(a / a-label : (b / b-label))
(a : (b))

# Cycles
```

(continues on next page)

(continued from previous page)

```
(a :ROLE (b :ROLE a))

# Distributed nodes
(a :ROLE (b :ROLE (c / c-label)) :ROLE2 (c :ATTR val))
```

Allowed but Unconventional

```
# Empty
()

# Missing edge target
(a / a-label :ROLE )

# Missing node label
(a / :ROLE (b / b-label))
```

Disallowed

```
# Disconnected (parseable as two separate graphs)
(a / a-label)(b / b-label)

# Missing identifiers
(a :ROLE ( / b-label))

# Misplaced label
(a :ROLE (b) / a-label)

# Multiple labels
(a / a-label / another-label)
```

CHAPTER

SIX

PENMAN

Penman graph library.

For basic usage, and to retain some backward compatibility with early versions, some functionality is available from the top-level `penman` module. For the rest, please use the standard API available via the submodules.

6.1 Submodules

6.1.1 Data Structures

- *penman.epigraph* – Base classes for epigraphical markers
- *penman.graph* – Classes for pure graphs
- *penman.model* – Class for defining semantic models
- *penman.models* – Pre-defined models
- *penman.surface* – Classes for surface alignments
- *penman.tree* – Classes for trees

6.1.2 Serialization

- *penman.codec* – Codec class for reading and writing PENMAN data
- *penman.layout* – Conversion between trees and graphs
- *penman.lexer* – Low-level parsing of PENMAN data

6.1.3 Other

- *penman.exceptions* – Exception classes
- *penman.interface* – Functional interface to a codec
- *penman.transform* – Graph and tree transformation functions

6.2 Module Constants

`penman.`**`__version__`**
The software version string.

`penman.`**`__version_info__`**
The software version as a tuple.

6.3 Classes

`class` `penman.`**`Triple`**
Alias of *`penman.graph.Triple`*.

`class` `penman.`**`Graph`**
Alias of *`penman.graph.Graph`*.

`class` `penman.`**`PENMANCodec`**
Alias of *`penman.codec.PENMANCodec`*.

6.4 Module Functions

`penman.`**`decode`**`()`
Alias of *`penman.interface.decode`*.

`penman.`**`loads`**`()`
Alias of *`penman.interface.loads`*.

`penman.`**`load`**`()`
Alias of *`penman.interface.load`*.

`penman.`**`encode`**`()`
Alias of *`penman.interface.encode`*.

`penman.`**`dumps`**`()`
Alias of *`penman.interface.dumps`*.

`penman.`**`dump`**`()`
Alias of *`penman.interface.dump`*.

6.5 Exceptions

`exception` `penman.`**`PenmanError`**
Alias of *`penman.exceptions.PenmanError`*.

`exception` `penman.`**`DecodeError`**
Alias of *`penman.exceptions.DecodeError`*.

CHAPTER

SEVEN

PENMAN.CODEC

Serialization of PENMAN graphs.

class penman.codec.**PENMANCodec**(*model=None*)

An encoder/decoder for PENMAN-serialized graphs.

ATOMS = {'FLOAT', 'INTEGER', 'STRING', 'SYMBOL'}

The valid non-node targets of edges.

decode(*s*, *triples=False*)

Deserialize PENMAN-notation string *s* into its Graph object.

Parameters

- **s** – a string containing a single PENMAN-serialized graph
- **triples** – if `True`, parse *s* as a triple conjunction

Returns The `Graph` object described by *s*.

Example

```
>>> codec = PENMANCodec()
>>> codec.decode('(b / bark :ARG1 (d / dog))')
<Graph object (top=b) at ...>
>>> codec.decode(
...     'instance(b, bark) ^ instance(d, dog) ^ ARG1(b, d)',
...     triples=True
... )
<Graph object (top=b) at ...>
```

iterdecode(*lines*, *triples=False*)

Yield graphs parsed from *lines*.

Parameters

- **lines** – a string or open file with PENMAN-serialized graphs
- **triples** – if `True`, parse *s* as a triple conjunction

Returns The `Graph` objects described in *lines*.

parse(*s*)

Parse PENMAN-notation string *s* into its tree structure.

Parameters **s** – a string containing a single PENMAN-serialized graph

Returns The tree structure described by *s*.

Example

```
>>> codec = PENMANCodec()
>>> codec.parse('(b / bark :ARG1 (d / dog))')  # noqa
Tree(('b', [('/', 'bark', []), ('ARG1', ('d', [('/', 'dog', [])]), [])]))
```

parse_triples (*s*)
Parse a triple conjunction from *s*.

encode (*g*, *top=None*, *triples=False*, *indent=-1*, *compact=False*)
Serialize the graph *g* into PENMAN notation.

Parameters

- **g** – the Graph object
- **top** – if given, the node to use as the top in serialization
- **triples** – if `True`, serialize as a conjunction of triples
- **indent** – how to indent formatted strings
- **compact** – if `True`, put initial attributes on the first line

Returns the PENMAN-serialized string of the Graph *g*

Example

```
>>> codec = PENMANCodec()
>>> codec.encode(Graph([('h', 'instance', 'hi')]))
(h / hi)
>>> codec.encode(Graph([('h', 'instance', 'hi')]),
...                          triples=True)
instance(h, hi)
```

format (*tree*, *indent=-1*, *compact=False*)
Format *tree* into a PENMAN string.

format_triples (*triples*, *indent=True*)
Return the formatted triple conjunction of *triples*.

Parameters

- **triples** – an iterable of triples
- **indent** – how to indent formatted strings

Returns the serialized triple conjunction of *triples*

Example

```
>>> codec = PENMANCodec()
>>> codec.format_triples([('a', ':instance', 'alpha'),
...                       ('a', ':ARG0', 'b'),
...                       ('b', ':instance', 'beta')])
...
'instance(a, alpha) ^\nARG0(a, b) ^\ninstance(b, beta)'
```

CHAPTER

EIGHT

PENMAN.EPIGRAPH

Base classes for epigraphical markers.

class `penman.epigraph.`**`Epidatum`**

`mode = 0`

The *`mode`* attribute specifies what the Epidatum annotates:

- `mode=0` – unspecified
- `mode=1` – role epidata
- `mode=2` – target epidata

CHAPTER

NINE

PENMAN.EXCEPTIONS

exception `penman.exceptions.`**`PenmanError`**

Base class for errors in the Penman package.

exception `penman.exceptions.`**`GraphError`**

Bases: *`penman.exceptions.PenmanError`*

Raises on invalid graph structures or operations.

exception `penman.exceptions.`**`LayoutError`**

Bases: *`penman.exceptions.PenmanError`*

Raised on invalid graph layouts.

exception `penman.exceptions.`**`DecodeError`**(*message=None*, *filename=None*, *lineno=None*, *offset=None*, *text=None*)

Bases: *`penman.exceptions.PenmanError`*

Raised on PENMAN syntax errors.

exception `penman.exceptions.`**`SurfaceError`**

Bases: *`penman.exceptions.PenmanError`*

Raised on invalid surface information.

exception `penman.exceptions.`**`ModelError`**

Bases: *`penman.exceptions.PenmanError`*

Raised when a graph violates model constraints.

CHAPTER

TEN

PENMAN.GRAPH

Data structures for Penman graphs and triples.

class penman.graph.**Graph**(*triples=None*, *top=None*, *epidata=None*, *metadata=None*)

A basic class for modeling a rooted, directed acyclic graph.

A Graph is defined by a list of triples, which can be divided into two parts: a list of graph edges where both the source and target are variables (node identifiers), and a list of node attributes where only the source is a variable and the target is a constant. The raw triples are available via the `triples` attribute, while the `edges()` and `attributes()` methods return only those that are edges between nodes or between a node and a constant, respectively.

Parameters

- **triples** – an iterable of triples (`Triple` or 3-tuples)
- **top** – the variable of the top node; if unspecified, the source of the first triple is used
- **epidata** – a mapping of triples to epigraphical markers
- **metadata** – a mapping of metadata types to descriptions

Example

```
>>> Graph([('b', ':instance', 'bark'),
...        ('d', ':instance', 'dog'),
...        ('b', ':ARG1', 'd')])
```

top

The top variable.

triples

The list of triples that make up the graph.

epidata

Epigraphical data that describe how a graph is to be expressed when serialized.

metadata

Metadata for the graph.

edges(*source=None*, *role=None*, *target=None*)

Return edges filtered by their *source*, *role*, or *target*.

Edges don't include terminal triples (concepts or attributes).

attributes(*source=None*, *role=None*, *target=None*)

Return attributes filtered by their *source*, *role*, or *target*.

Attributes don't include triples where the target is a nonterminal.

variables()

Return the set of variables (nonterminal node identifiers).

reentrancies()

Return a mapping of variables to their re-entrancy count.

A re-entrancy is when more than one edge selects a node as its target. These graphs are rooted, so the top node always has an implicit entrancy. Only nodes with re-entrancies are reported, and the count is only for the entrant edges beyond the first. Also note that these counts are for the interpreted graph, not for the linearized form, so inverted edges are always re-entrant.

class penman.graph.**Triple**

A relation between nodes or between a node and an constant.

Parameters

- **source** – the source variable of the triple
- **role** – the edge label between the source and target
- **target** – the target variable or constant

source

The source variable of the triple.

role

The edge label between the source and target.

target

The target variable or constant.

class penman.graph.**Edge**

Bases: *penman.graph.Triple*

A relation between nodes.

class penman.graph.**Attribute**

Bases: *penman.graph.Triple*

A relation between a node and a constant.

CHAPTER

ELEVEN

PENMAN.INTERFACE

Functions for basic reading and writing of PENMAN graphs.

11.1 Graph-reading Functions

`penman.interface.`**`decode`**(*s*, *model=None*, *triples=False*)

Deserialize PENMAN-serialized *s* into its Graph object

Parameters

- **`s`** – a string containing a single PENMAN-serialized graph
- **`model`** – the model used for interpreting the graph
- **`triples`** – if `True`, read as a conjunction of triples

Returns the Graph object described by *s*

Example

```
>>> decode('(b / bark :ARG1 (d / dog))')
<Graph object (top=b) at ...>
```

`penman.interface.`**`loads`**(*string*, *model=None*, *triples=False*)

Deserialize a list of PENMAN-encoded graphs from *string*.

Parameters

- **`string`** – a string containing graph data
- **`model`** – the model used for interpreting the graph
- **`triples`** – if `True`, read as a conjunction of triples

Returns a list of Graph objects

`penman.interface.`**`load`**(*source*, *model=None*, *triples=False*)

Deserialize a list of PENMAN-encoded graphs from *source*.

Parameters

- **`source`** – a filename or file-like object to read from
- **`model`** – the model used for interpreting the graph
- **`triples`** – if `True`, read as a conjunction of triples

Returns a list of Graph objects

11.2 Graph-writing Functions

`penman.interface.`**`encode`**(*g*, *top=None*, *model=None*, *triples=False*, *indent=-1*, *compact=False*)

Serialize the graph *g* from *top* to PENMAN notation.

Parameters

- **`g`** – the Graph object
- **`top`** – if given, the node to use as the top in serialization
- **`model`** – the model used for interpreting the graph
- **`triples`** – if `True`, serialize as a conjunction of triples
- **`indent`** – how to indent formatted strings
- **`compact`** – if `True`, put initial attributes on the first line

Returns the PENMAN-serialized string of the Graph *g*

Example

```
>>> encode(Graph([('h', 'instance', 'hi')]))
(h / hi)
```

`penman.interface.`**`dumps`**(*graphs*, *model=None*, *triples=False*, *indent=-1*, *compact=False*)

Serialize each graph in *graphs* to the PENMAN format.

Parameters

- **`graphs`** – an iterable of Graph objects
- **`model`** – the model used for interpreting the graph
- **`triples`** – if `True`, serialize as a conjunction of triples
- **`indent`** – how to indent formatted strings
- **`compact`** – if `True`, put initial attributes on the first line

Returns the string of serialized graphs

`penman.interface.`**`dump`**(*graphs*, *file*, *model=None*, *triples=False*, *indent=-1*, *compact=False*)

Serialize each graph in *graphs* to PENMAN and write to *file*.

Parameters

- **`graphs`** – an iterable of Graph objects
- **`file`** – a filename or file-like object to write to
- **`model`** – the model used for interpreting the graph
- **`triples`** – if `True`, serialize as a conjunction of triples
- **`indent`** – how to indent formatted strings
- **`compact`** – if `True`, put initial attributes on the first line

CHAPTER

TWELVE

PENMAN.LAYOUT

Interpreting trees to graphs and configuring graphs to trees.

In order to serialize graphs into the PENMAN format, a tree-like layout of the graph must be decided. Deciding a layout includes choosing the order of the edges from a node and the paths to get to a node definition (the position in the tree where a node's concept and edges are specified). For instance, the following graphs for "The dog barked loudly" have different edge orders on the `b` node:

```
(b / bark-01                (b / bark-01
   :ARG0 (d / dog)             :mod (l / loud)
   :mod (l / loud))            :ARG0 (d / dog))
```

With re-entrancies, there are choices about which location of a re-entrant node gets the full definition with its concept (node label), etc. For instance, the following graphs for "The dog tried to bark" have different locations for the definition of the `d` node:

```
(t / try-01                   (t / try-01
   :ARG0 (d / dog)               :ARG0 d
   :ARG1 (b / bark-01            :ARG1 (b / bark-01
      :ARG0 d))                     :ARG0 (d / dog))
```

With inverted edges, there are even more possibilities, such as:

```
(t / try-01                     (t / try-01
   :ARG0 (d / dog                  :ARG1 (b / bark-01
      :ARG0-of b)                     :ARG0 (d / dog
   :ARG1 (b / bark-01))                  :ARG0-of t)))
```

This module introduces two epigraphical markers so that a pure graph parsed from PENMAN can retain information about its tree layout without altering its graph properties. The first marker type is *Push*, which is put on a triple to indicate that the triple introduces a new node context, while the sentinel *POP* indicates that a triple is at the end of one or more node contexts. These markers only work if the triples in the graph's data are ordered. For instance, one of the graphs above (repeated here) has the following data:

```
PENMAN                  Graph                                Epigraph
(t / try-01             [('t', ':instance', 'try-01'),       :
   :ARG0 (d / dog)       ('t', ':ARG0', 'd'),                : Push('d')
   :ARG1 (b / bark-01    ('d', ':instance', 'dog'),          : POP
      :ARG0 d))          ('t', ':ARG1', 'b'),                : Push('b')
                         ('b', ':instance', 'bark-01'),      :
                         ('b', ':ARG0', 'd')]                : POP
```

12.1 Epigraphical Markers

class penman.layout.**LayoutMarker**

Bases: *penman.epigraph.Epidatum*

Epigraph marker for layout choices.

class penman.layout.**Push**(*variable*)

Bases: *penman.layout.LayoutMarker*

Epigraph marker to indicate a new node context.

penman.layout.**POP = POP**

Epigraphical marker to indicate the end of a node context.

12.2 Tree Functions

penman.layout.**interpret**(*t*, *model=None*)

Interpret tree *t* as a graph using *model*.

Tree interpretation is the process of transforming the nodes and edges of a tree into a directed graph. A semantic model determines which edges are inverted and how to deinvert them. If *model* is not provided, the default model will be used.

Parameters

- **t** – the `Tree` to interpret
- **model** – the *Model* used to interpret *t*

Returns The interpreted *Graph*.

Example

```
>>> from penman.tree import Tree
>>> from penman import layout
>>> t = Tree(
...   ('b', [
...     ('/', 'bark', []),
...     ('ARG0', ('d', [
...       ('/', 'dog', [])]), [])]))
>>> g = layout.interpret(t)
>>> for triple in g.triples:
...     print(triple)
...
('b', ':instance', 'bark')
('b', ':ARG0', 'd')
('d', ':instance', 'dog')
```

penman.layout.**rearrange**(*t*, *key=None*)

Sort the branches at each node in tree *t* according to *key*.

Each node in a tree contains a list of branches. This function sorts those lists in-place using the *key* function, which accepts a branch and returns some sortable criterion. If the first branch is the node label it will stay in place after the sort.

Example

```
>>> from penman import layout
>>> from penman.model import Model
>>> from penman.codec import PENMANCodec
>>> c = PENMANCodec()
>>> t = c.parse('(s / see :ARG0 (d / dog) :ARG1 (c / cat))')
>>> layout.rearrange(t, key=Model().random_order)
>>> print(c.format(t))
(s / see
   :ARG1 (c / cat)
   :ARG0 (d / dog))
```

12.3 Graph Functions

`penman.layout.`**`configure`**`(`*g*, *top=None*, *model=None*, *strict=False*`)`

Create a tree from a graph by making as few decisions as possible.

A graph interpreted from a valid tree using *`interpret()`* will contain epigraphical markers that describe how the triples of a graph are to be expressed in a tree, and thus configuring this tree requires only a single pass through the list of triples. If the markers are missing or out of order, or if the graph has been modified, then the configuration process will have to make decisions about where to insert tree branches. These decisions are deterministic, but may result in a tree different than the one expected.

Parameters

- **`g`** – the *`Graph`* to configure
- **`top`** – the variable to use as the top of the graph; if `None`, the top of *g* will be used
- **`model`** – the *`Model`* used to configure the tree
- **`strict`** – if `True`, raise *`LayoutError`* if decisions must be made about the configuration

Returns The configured `Tree`.

Example

```
>>> from penman.graph import Graph
>>> from penman import layout
>>> g = Graph([('b', ':instance', 'bark'),
...            ('b', ':ARG0', 'd'),
...            ('d', ':instance', 'dog')])
>>> t = layout.configure(g)
>>> print(t)
Tree(
  ('b', [
    ('/', 'bark', []),
    (':ARG0', ('d', [
      ('/', 'dog', [])]), [])]))
```

`penman.layout.`**`reconfigure`**`(`*g*, *top=None*, *model=None*, *strict=False*`)`

Create a tree from a graph after any discarding layout markers.

12.4 Diagnostic Functions

`penman.layout.`**`has_valid_layout`**`(`*g*, *top=None*, *model=None*, *strict=False*`)`

Return `True` if *g* contains the information for a valid layout.

Having a valid layout means that the graph data allows a depth-first traversal that reconstructs a spanning tree used for serialization.

`penman.layout.`**`appears_inverted`**`(`*g*, *triple*`)`

Return `True` if *triple* appears inverted in serialization.

More specifically, this function returns `True` if *triple* has a *`Push`* epigraphical marker in graph *g* whose associated variable is the source variable of *triple*. This should be accurate when testing a triple in a graph interpreted using *`interpret()`* (including *`PENMANCodec.decode`*, etc.), but it does not guarantee that a new serialization of *g* will express *triple* as inverted as it can change if the graph or its epigraphical markers are modified, if a new top is chosen, etc.

Parameters

- **`g`** – a *`Graph`* containing *triple*
- **`triple`** – the triple that does or does not appear inverted

Returns `True` if *triple* appears inverted in graph *g*.

CHAPTER

THIRTEEN

PENMAN.LEXER

Classes and functions for lexing PENMAN strings.

13.1 Module Constants

penman.lexer.**PATTERNS**

A dictionary mapping token names to regular expressions. For instance:

```
'ROLE':  r':[^\s()\/,:~^]*'
```

The token names are used later by the `TokenIterator` to help with parsing.

penman.lexer.**PENMAN_RE**

A compiled regular expression pattern for lexing PENMAN graphs.

penman.lexer.**TRIPLE_RE**

A compiled regular expression pattern for lexing triple conjunctions.

13.2 Module Functions

penman.lexer.**lex**(*lines*, *pattern=None*)

Yield PENMAN tokens matched in *lines*.

By default, this lexes strings in *lines* using the basic pattern for PENMAN graphs. If *pattern* is given, it is used for lexing instead.

Parameters

- **lines** – iterable of lines to lex
- **pattern** – pattern to use for lexing instead of the default ones

Returns A `TokenIterator` object

13.3 Classes

class penman.lexer.**Token**

A lexed token.

property line

The line the token appears in.

property lineno
The line number the token appears on.

property offset
The character offset of the token.

property text
The matched string for the token.

property type
The token type.

class `penman.lexer.`**TokenIterator**(*iterator*)
An iterator of Tokens with L1 lookahead.

accept(**choices*)
Return the next token if its type is in *choices*.

The iterator is advanced if successful. If unsuccessful, `None` is returned.

expect(**choices*)
Return the next token if its type is in *choices*.

The iterator is advanced if successful.

Raises `DecodeError` – if the next token type is not in *choices*

next()
Advance the iterator and return the next token.

Raises `StopIteration` – if the iterator is already exhausted.

peek()
Return the next token but do not advance the iterator.

If the iterator is exhausted then a `DecodeError` is raised.

PENMAN.MODEL

Semantic models for interpreting graphs.

class penman.model.**Model**(*top_variable='top'*, *top_role=':TOP'*, *concept_role=':instance'*, *roles=None*, *normalizations=None*, *reifications=None*)

A semantic model for Penman graphs.

The model defines things like valid roles and transformations.

Parameters

- **top_variable** – the variable of the graph's top
- **top_role** – the role linking the graph's top to the top node
- **concept_role** – the role associated with node concepts
- **roles** – a mapping of roles to associated data
- **normalizations** – a mapping of roles to normalized roles
- **reifications** – a list of 4-tuples used to define reifications

classmethod from_dict(*d*)

Instantiate a model from a dictionary.

has_role(*role*)

Return `True` if *role* is defined by the model.

If *role* is not in the model but a single deinversion of *role* is in the model, then `True` is returned. Otherwise `False` is returned, even if something like `canonicalize_role()` could return a valid role.

is_role_inverted(*role*)

Return `True` if *role* is inverted.

invert_role(*role*)

Invert *role*.

invert(*triple*)

Invert *triple*.

This will invert or deinvert a triple regardless of its current state. `deinvert()` will deinvert a triple only if it is already inverted. Unlike `canonicalize()`, this will not perform multiple inversions or replace the role with a normalized form.

deinvert(*triple*)

De-invert *triple* if it is inverted.

Unlike `invert()`, this only inverts a triple if the model considers it to be already inverted, otherwise it is left alone. Unlike `canonicalize()`, this will not normalize multiple inversions or replace the role with a normalized form.

`canonicalize_role`(*role*)
Canonicalize *role*.

Role canonicalization will do the following:

- Ensure the role starts with *':'*
- Normalize multiple inversions (e.g., `ARG0-of-of` becomes `ARG0`), but it does *not* change the direction of the role
- Replace the resulting role with a normalized form if one is defined in the model

`canonicalize`(*triple*)
Canonicalize *triple*.

See *`canonicalize_role()`* for a description of how the role is canonicalized. Unlike *`invert()`*, this does not swap the source and target of *triple*.

`is_reifiable`(*triple*)
Return `True` if the role of *triple* can be reified.

`reify`(*triple*, *variables=None*)
Return the three triples that reify *triple*.

Note that, unless *variables* is given, the node variable for the reified node is not necessarily valid for the target graph. When incorporating the reified triples, this variable should then be replaced.

If the role of *triple* does not have a defined reification, a `ModelError` is raised.

Parameters

- **`triple`** – the triple to reify
- **`variables`** – a set of variables that should not be used for the reified node's variable

Returns The 3-tuple of triples that reify *triple*.

`original_order`(*branch*)
Branch sorting key that does not change the order.

`canonical_order`(*branch*)
Branch sorting key that finds a canonical order.

`random_order`(*branch*)
Branch sorting key that randomizes the order.

CHAPTER

FIFTEEN

PENMAN.MODELS

This sub-package contains specified instances of the `penman.model.Model` class, although currently there is only one instance.

15.1 Available Models

15.1.1 penman.models.amr

AMR semantic model definition.

`penman.models.amr.`**`model = <penman.model.Model object>`**

The AMR model is an instance of `Model` using the roles, normalizations, and reifications defined in this module.

`penman.models.amr.`**`roles = {':ARG0': {'type': 'frame'}, ':ARG1': {'type': 'frame'}, ':A`**

The roles are the edge labels of reifications. The purpose of roles in a `Model` is mainly to define the set of valid roles, but they map to arbitrary data which is not used by the `Model` but may be inspected or used by client code.

`penman.models.amr.`**`normalizations = {':domain-of': ':mod', ':mod-of': ':domain'}`**

Normalizations are like role aliases. If the left side of the normalization is encountered by `Model.canonicalize_role()` then it is replaced with the right side.

`penman.models.amr.`**`reifications = [(':accompanier', 'accompany-01', ':ARG0', ':ARG1'), (':a`**

Reifications are a particular kind of transformation that replaces an edge relation with a new node and two outgoing edge relations, one inverted. They are used when the edge needs to behave as a node, e.g., to be modified or focused.

CHAPTER

SIXTEEN

PENMAN.SURFACE

Surface strings, tokens, and alignments.

16.1 Epigraphical Markers

class penman.surface.**AlignmentMarker**(*indices*, *prefix=None*)
    Bases: *penman.epigraph.Epidatum*

class penman.surface.**Alignment**(*indices*, *prefix=None*)
    Bases: *penman.surface.AlignmentMarker*

class penman.surface.**RoleAlignment**(*indices*, *prefix=None*)
    Bases: *penman.surface.AlignmentMarker*

16.2 Module Functions

penman.surface.**alignments**(*g*)
    Return a mapping of triples to alignments in graph *g*.

Parameters **g** – a Graph containing alignment data

Returns A dict mapping Triple objects to their corresponding *Alignment* objects, if any.

penman.surface.**role_alignments**(*g*)
    Return a mapping of triples to role alignments in graph *g*.

Parameters **g** – a Graph containing role alignment data

Returns A dict mapping Triple objects to their corresponding *RoleAlignment* objects, if any.

CHAPTER

SEVENTEEN

PENMAN.TRANSFORM

Tree and graph transformations.

`penman.transform.`**`canonicalize_roles`**`(`*t*, *model*`)`

Normalize roles in *t* so they are canonical according to *model*.

This is a tree transformation instead of a graph transformation because the orientation of the pure graph's triples is not decided until the graph is configured into a tree.

Parameters

- **`t`** – a `Tree` object
- **`model`** – a model defining role normalizations

Returns A new `Tree` object with canonicalized roles.

Example

```
>>> from penman.codec import PENMANCodec
>>> from penman.models.amr import model
>>> from penman.transform import canonicalize_roles
>>> codec = PENMANCodec()
>>> t = codec.parse('(c / chapter :domain-of 7)')
>>> t = canonicalize_roles(t, model)
>>> print(codec.format(t))
(c / chapter
   :mod 7)
```

`penman.transform.`**`reify_edges`**`(`*g*, *model*`)`

Reify all edges in *g* that have reifications in *model*.

Parameters

- **`g`** – a `Graph` object
- **`model`** – a model defining reifications

Returns A new `Graph` object with reified edges.

Example

```
>>> from penman.codec import PENMANCodec
>>> from penman.models.amr import model
>>> from penman.transform import reify_edges
```

(continues on next page)

(continued from previous page)

```
>>> codec = PENMANCodec(model=model)
>>> g = codec.decode('(c / chapter :mod 7)')
>>> g = reify_edges(g, model)
>>> print(codec.encode(g))
(c / chapter
   :ARG1-of (_ / have-mod-91
               :ARG2 7))
```

penman.transform.**reify_attributes**(*g*)

Reify all attributes in *g*.

Parameters **g** – a `Graph` object

Returns A new `Graph` object with reified attributes.

Example

```
>>> from penman.codec import PENMANCodec
>>> from penman.models.amr import model
>>> from penman.transform import reify_attributes
>>> codec = PENMANCodec(model=model)
>>> g = codec.decode('(c / chapter :mod 7)')
>>> g = reify_attributes(g)
>>> print(codec.encode(g))
(c / chapter
   :mod (_ / 7))
```

penman.transform.**indicate_branches**(*g*, *model*)

Insert TOP triples in *g* indicating the tree structure.

Note: This depends on *g* containing the epigraphical layout markers from parsing; it will not work with programmatically constructed Graph objects or those whose epigraphical data were removed.

Parameters

- **g** – a `Graph` object
- **model** – a model defining the TOP role

Returns A new `Graph` object with TOP roles indicating tree branches.

Example

```
>>> from penman.codec import PENMANCodec
>>> from penman.models.amr import model
>>> from penman.transform import indicate_branches
>>> codec = PENMANCodec(model=model)
>>> g = codec.decode('''
... (w / want-01
...    :ARG0 (b / boy)
...    :ARG1 (g / go-02
...             :ARG0 b))''')
>>> g = indicate_branches(g, model)
```

(continues on next page)

(continued from previous page)

```
>>> print(codec.encode(g))
(w / want-01
   :TOP b
   :ARG0 (b / boy)
   :TOP g
   :ARG1 (g / go-02
            :ARG0 b))
```

CHAPTER EIGHTEEN

PENMAN.TREE

Definitions of tree structures.

class penman.tree.**Tree**(*node*, *metadata=None*)

A tree structure.

A tree is essentially a node that contains other nodes, but this Tree class is useful to contain any metadata and to provide tree-based methods.

nodes()

Return the nodes in the tree as a flat list.

penman.tree.**is_atomic**(*x*)

Return `True` if *x* is a valid atomic value.

CHAPTER NINETEEN

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[KAS1989] Robert T. Kaspar. A Flexible Interface for Linking Applications to Penman's Sentence Generator. Speech and Natural Language: Proceedings of a Workshop Held at Philadelphia, Pennsylvania. http://www.aclweb.org/anthology/H89-1022. February 21-23, 1989.

[GOO2019] Michael Wayne Goodman. AMR Normalization for Fairer Evaluation. Proceedings of the 33rd Pacific Asia Conference on Language, Information, and Computation (PACLIC 33). https://arxiv.org/pdf/1909.01568.pdf. 2019.

PYTHON MODULE INDEX

INDEX