Penman Documentation

Release 1.1.0

Michael Wayne Goodman

Jul 06, 2020

GUIDES

Quick Links

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

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

USING THE PENMAN COMMAND

The **penman** command allows users to perform most reformatting tasks and predefined transformations without having to write any code. For example, the following reformats a graph in one line to a more conventional presentation:

```
$ penman --indent 3 --compact <<< '(s / sleep-01 :polarity - :ARG0 (i / i))'
(s / sleep-01 :polarity -
   :ARG0 (i / i))
```

The command becomes available at the command-line after installing Penman (see *Installation and Setup*). This guide will explain how to use the command for several use cases.

Contents

2.1 Command Usage

The **penman** command reads in data from stdin or from one or more files then prints the results to stdout. By default, the command will do nothing but apply the default formatting to the graphs, but any input content that is not a graph or a metadata comment will be discarded. To see what features are available for the current version and how to call the command, run **penman --help**:

```
usage: penman [-h] [-V] [-v] [-q] [--model FILE | --amr | --noop] [--check]
              [--indent N] [--compact] [--triples] [--make-variables FMT]
              [--rearrange KEY] [--reconfigure KEY] [--canonicalize-roles]
              [--reify-edges] [--dereify-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
  --noop                use the no-op model
  --check               check graphs for compliance with the model

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:
  --make-variables FMT  recreate node variables with FMT (e.g.: '{prefix}{j}')
  --rearrange KEY       reorder the branches of the tree
  --reconfigure KEY     reconfigure the graph layout with reordered triples
  --canonicalize-roles  canonicalize role forms
  --reify-edges         reify all eligible edges
  --dereify-edges       dereify all eligible edges
  --reify-attributes    reify all attributes
  --indicate-branches   insert triples to indicate tree structure
```

2.2 Reformatting

There are two options for reformatting graphs that use newlines and indentation to make them more friendly to human eyes. The **--indent** option controls how much each nesting level indents and the **--compact** option determines whether attributes immediately following a concept appear on the same line as the concept or on their own lines. For this section, consider the following graph:

```
$ x="(w / want-01 :polarity - :ARG0 (c / child) :ARG1 (g / go :ARG0 c))"
```

2.2.1 Default Formatting

By default, compact mode is off and **`--indent`** has the special value `-1`, which performs "adaptive indenting". This appears as follows:

```
$ echo "$x" | penman
(w / want-01
   :polarity -
   :ARG0 (c / child)
   :ARG1 (g / go
            :ARG0 c))
```

2.2.2 Changing the Indentation

Giving a specific indent number makes Penman always indent that number of spaces:

```
$ echo "$x" | penman --indent 3
(w / want-01
   :polarity -
   :ARG0 (c / child)
   :ARG1 (g / go
      :ARG0 c))
```

2.2.3 Compact Attributes

Compact mode puts attributes on the same line as the concept of their node, but only if they appear in that position in the tree:

```
$ echo "$x" | penman --compact
(w / want-01 :polarity -
   :ARG0 (c / child)
   :ARG1 (g / go
            :ARG0 c))
```

2.2.4 Single-Line Graphs

With **`--indent=no`**, Penman outputs a full graph on one line. This can be useful for programs that read data line-by-line or for creating bilingually aligned files:

```
$ echo "$x" | penman
(w / want-01
   :polarity -
   :ARG0 (c / child)
   :ARG1 (g / go
            :ARG0 c))
$ echo "$x" | penman | penman --indent=no
(w / want-01 :polarity - :ARG0 (c / child) :ARG1 (g / go :ARG0 c))
```

Note that **`--indent=0`** is not the same as **`--indent=no`**. The former delimits parts with a single newline but no leading space whereas the latter delimits parts with a single space and no newlines. Also, the **`--compact`** option is relevant when **`--indent`** has a numeric value but not for **`--indent=no`**.

2.3 Specifying a Model

While the formatting options do not require knowledge of the semantic model, others, such as **`--check`** and many transformations, do require it. For Abstract Meaning Representation (AMR) graphs, the **`--amr`** option uses the built-in AMR model:

```
$ penman --amr [...]
```

This model contains information about AMR's valid roles, canonical role inversions (such as `:domain` to `:mod`), and relation reifications. Also available is the no-op model via **`--noop`**, which does not deinvert tree edges when interpreting the graph so that a role like `:ARG0-of` is the role used in the graph triples.

Other models can be given by using the **`--model`** option with a path to a JSON file containing the model information:

```
$ penman --model=xyz.json [...]
```

Custom models can be used for variations of AMR (e.g., different versions or task-specific definitions) or even for different semantic frameworks altogether.

2.4 Checking for Model Compliance

With a model specified, a graph can be checked for compliance with respect to the model using the **`--check`** option. For graphs already in PENMAN notation, the only relevant test is whether a role is defined by the model. When graphs are constructed programatically, there are additional checks for graphical well-formedness, such as for an appropriate graph-top being set and for graph connectedness. When used as a command, the exit code of the command will be `0` when there are no errors or `1` when any errors are found. This helps make the check be scriptable. Also, the individual errors are inserted as metadata comments on each graph to help users resolve errors:

```
$ good="(s / swim-01 :ARG0 (i / i))"                          # I swim.
$ bad="(s / swim-01 :ARG0 (i / i) :stroke (b / backstroke))"  # I swim backstroke.
$ if ( echo "$good" | penman --amr --check ); then
>   echo "valid"
> else
>   echo "invalid"
> fi
(s / swim-01
   :ARG0 (i / i))
valid
$ if ( echo "$bad" | penman --amr --check ); then
>   echo "valid"
> else
>   echo "invalid"
> fi
# ::error-1 (s :stroke b) invalid role
(s / swim-01
   :ARG0 (i / i)
   :stroke (b / backstroke))
invalid
```

2.5 Transforming Graphs

Penman's transformations work either on the tree or the graph representation.

2.5.1 Relabeling Nodes

The simplest transformation maps variables to a new form with the **`--make-variables`** option. In English AMR the variables use the first letter of the concept and, if it is not unique, the 1-based index starting from the second when traversing the tree in depth-first order. AMR's primary evaluation tool smatch relabels all nodes internally so one side uses `a0`, `a1`, etc. and the other side uses `b0`, `b1`, etc. Penman allows users to specify the variable format with three template variables:

- `{prefix}` uses the first character of a node's concept
- `{i}` is the 0-based index of a node's occurrence
- `{j}` is the 1-based index of a node's occurrence, where index 1 is blank

Unlike the other transformations, **`--make-variables`** does not require a model:

```
$ original="(x0 / chase-01 :ARG0 (x1 / cat) :ARG1 (x2 / mouse))"
$ echo "$original" | penman --make-variables='a{i}'
(a0 / chase-01
    :ARG0 (a1 / cat)
    :ARG1 (a2 / mouse))
$ echo "$original" | penman --make-variables='{prefix}{j}'
(c / chase-01
   :ARG0 (c2 / cat)
   :ARG1 (m / mouse))
```

2.5.2 Rearranging Branches

Tree branches can be rearranged without changing the overall tree structure using the **`--rearrange`** option. It takes the name of a method for sorting the branches on a node:

```
$ original="(c / chase-01 :ARG1 (m / mouse) :polarity - :ARG0 (c2 / cat))"
$ echo "$original" | penman --rearrange=attributes-first
(c / chase-01
   :polarity -
   :ARG1 (m / mouse)
   :ARG0 (c2 / cat))
$ echo "$original" | penman --rearrange=alphanumeric
(c / chase-01
   :ARG0 (c2 / cat)
   :ARG1 (m / mouse)
   :polarity -)
```

The sorting methods can be combined in prioritized order:

```
$ echo "$original" | penman --rearrange=attributes-first,alphanumeric
(c / chase-01
   :polarity -
   :ARG0 (c2 / cat)
   :ARG1 (m / mouse))
```

2.5.3 Reconfiguring the Tree

In Penman, the *epigraph* is a side-channel of information that allows it to configure (reconstruct) the original tree that led to a graph representation. The **`--reconfigure`** option first discards this epigraphical information then configures the tree afresh, which may lead to more drastic restructuring than just rearranging tree branches. Like **`--rearrange`**, it takes a sorting method as its argument. Often it is helpful to use **`--rearrange`** with **`--reconfigure`**, so the reconfigured tree still follows an expected branch order:

```
$ original="(s / sell-01 :ARG0 (i / i) :ARG1 (b / book :ARG1-of (r / read :ARG0 i)))"
$ echo "$original" | penman
(s / sell-01
 :ARG0 (i / i)
 :ARG1 (b / book
          :ARG1-of (r / read
                      :ARG0 i)))
$ echo "$original" | penman --reconfigure=random --rearrange=alphanumeric
(s / sell-01
   :ARG0 (i / i
            :ARG0-of (r / read
                        :ARG1 (b / book)))
   :ARG1 b)
```

Note that **`--reconfigure`** does not change which variable is the graph's top. This is because the resulting graph should encode the same information, and the top node is treated specially. For example, in AMR it is considered the *focused* node. A reconfigured graph will return a perfect score with the original using a metric like smatch.

2.5.4 Normalizations

The remaining options are normalizations that may alter the content of the graph. The **`--canonicalize-roles`** option will replace roles that the model defines as equivalent, such as `:domain-of` and `:mod` in AMR:

```
$ echo "(c / chapter :domain-of 7)" | penman --amr --canonicalize-roles
(c / chapter
   :mod 7)
```

Penman can handle relations that are over-inverted one time, but does not check further than that. The **`--canonicalize-roles`** option will try harder to resolve over-inversions. For this functionality, a model is not strictly necessary unless the over-inverted role itself needs to be canonicalized:

```
$ echo "(b / bark-01 :ARG0-of-of (d / dog))" | penman
(b / bark-01
   :ARG0 (d / dog))
$ echo "(b / bark-01 :ARG0-of-of-of-of (d / dog))" | penman
(b / bark-01
   :ARG0-of-of (d / dog))
$ echo "(b / bark-01 :ARG0-of-of-of-of (d / dog))" | penman --canonicalize-roles
(b / bark-01
   :ARG0 (d / dog))
```

The **`--reify-edges`** option converts edges into nodes for edges that have a reification defined in the model:

```
$ echo "(c / chapter :mod 7)" | penman --amr --reify-edges
(c / chapter
   :ARG1-of (_ / have-mod-91
               :ARG2 7))
```

The _ (_2, etc.) variables indicate which have been reified. Combine with **--make-variables** to use standard variable names (e.g., h in this example). The **--dereify-edges** is the reverse of **--reify-edges**:

```
$ echo "(c / chapter :mod 7)" | penman --amr --reify-edges | penman --amr --dereify-
↪edges
(c / chapter
   :mod 7)
```

The **--reify-attributes** option reifies attribute relations (those where the value is a constant) so the constant value becomes the concept of a new node:

```
$ echo "(c / chapter :mod 7)" | penman --amr --reify-attributes
(c / chapter
   :mod (_ / 7))
```

Finally, the **--indicate-branches** option inserts relations that hint at the original tree structure. This can be useful if a tool that produces PENMAN graphs, like an AMR parser, wants to use a tool like smatch to compare its output to gold trees and not just gold graphs.

CHAPTER

THREE

USING PENMAN AS A PYTHON LIBRARY

For some cases, the **penman** command is not flexible enough and it becomes necessary to write some Python code. Penman's Python API is well-documented and well-tested and lets you dig into the actual structures holding the data. One case where it's currently necessary to write code is for arbitrary graph editing. For example, perhaps you want to anonymize all attributes with numeric values. Here is one way to do that with the API:

```
>>> import penman
>>> from penman import constant
>>> g = penman.decode('(b / buy-01 :ARG0 (i / i) :ARG1 (a / apple :quant 3))')
>>> anon_map = {}
>>> attributes = []
>>> for src, role, tgt in g.attributes():
...     if constant.type(tgt) in (constant.INTEGER, constant.FLOAT):
...         anon_val = f'number_{len(anon_map)}'
...         anon_map[anon_val] = tgt
...         tgt = anon_val
...     attributes.append((src, role, tgt))
...
>>> g2 = penman.Graph(g.instances() + g.edges() + attributes)
>>> print(penman.encode(g2))
(b / buy-01
   :ARG0 (i / i)
   :ARG1 (a / apple
            :quant number_0))
>>> anon_map
{'number_0': '3'}
```

This could be improved, such as making the anonymization into a function. It could also be made to work on the *Tree* structure if you care about keeping the original tree intact as this procedure loses the epigraphical markers needed to reconstruct the tree from the graph.

The API is also useful for deeper inspection of graphs. For example:

```
>>> import penman
>>> g = penman.decode('''
... # ::id ex1 ::snt The dog barked.
... (b / bark-01
...    :ARG0 (d / dog))
... ''')
>>> g.top
'b'
>>> g.instances()
[Instance(source='b', role=':instance', target='bark-01'), Instance(source='d', role=
↪':instance', target='dog')]
>>> g.edges()
```

(continues on next page)

(continued from previous page)

```
[Edge(source='b', role=':ARG0', target='d')]
>>> sorted(g.variables())
['b', 'd']
>>> g.metadata
{'snt': 'The dog barked.', 'id': 'ex1'}
>>> g.epidata
{('b', ':instance', 'bark-01'): [], ('b', ':ARG0', 'd'): [Push(d)], ('d', ':instance',
↪ 'dog'): [POP]}
>>> g.reentrancies()
{}
```

Or for inserting surface alignments:

```
>>> from penman import surface
>>> g.metadata['tok'] = 'The dog barked .'
>>> g.epidata[('b', ':instance', 'bark-01')].append(surface.Alignment((2,), prefix='e.
↪'))
>>> g.epidata[('d', ':instance', 'dog')].append(surface.Alignment((1,), prefix='e.'))
>>> print(penman.encode(g))
# ::snt The dog barked.
# ::id ex1
# ::tok The dog barked .
(b / bark-01~e.2
   :ARG0 (d / dog~e.1))
```

Many tasks can be accomplished with the basic API available at the top-level *penman* module, but some more advanced usage requires the use of specific submodules, such as the use of *penman.constant* and *penman.surface* above. See the *API documentation* for more information.

CHAPTER

FOUR

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 -))
```

Described below are a breakdown of the parts of the PENMAN graph above as well as a formal grammar description of PENMAN graphs in general.

4.1 Graph Anatomy

The following diagram explains what each part of the graph above is:

```
;    ┌──────────────────────────── Variable (this one is the graph's top)
;    │     ┌─────────────────────── Instance relation
;     ┌─────────┐
    (d / drive-01
;       ┌──────┐
;       │    └──────────────────── Concept (node label)
;       └──────────────────────── Indicates the node's concept
;            ┌─────────────────── 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.

4.2 Formal Grammar

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

# Lexical productions (whitespace is not allowed)
Symbol    <- NameChar+
Role      <- ':' NameChar*
Alignment <- '~' ([a-zA-Z] '.'?)? Digit+ (',' Digit+)*
String    <- '"' (!'"' ('\\' . / .))* '"'
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.

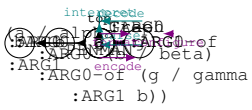
Note that the implementation in the Penman package deviates from this grammar in that the `Alignment` production is not parsed together with the rest of the structure. Instead, the ~ character is allowed on `NameChar` and alignments are thus part of the `Role` or `Atom` tokens. They are later detected and extracted during graph interpretation (see *`penman.layout.interpret()`*).

CHAPTER

FIVE

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 (concepts are not shown on the tree and graph for simplicity):



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

SIX

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-01
   :ARG0 (d / dog))
```

The above PENMAN string is parsed to the following tree:

```
Tree(('b', [(':instance', 'bark-01'),
            (':ARG0', ('d', [(':instance', 'dog')]))]))
```

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

```
Graph(triples=[
       ('b', ':instance', 'bark-01'),
       ('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.

6.1 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))

# Inverted attributes
(a / a-label :ARG0-of 2)
```

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

SEVEN

PENMAN

Penman graph library.

For basic usage, common functionality is available from the top-level *penman* module. For more advanced usage, please use the full API available via the submodules.

Users wanting to interact with graphs might find the *decode()* and *encode()* functions a good place to start:

```
>>> import penman
>>> g = penman.decode('(w / want-01 :ARG0 (b / boy) :ARG1 (g / go :ARG0 b))')
>>> g.top
'w'
>>> len(g.triples)
6
>>> [concept for _, _, concept in g.instances()]
['want-01', 'boy', 'go']
>>> print(penman.encode(g, top='b'))
(b / boy
   :ARG0-of (w / want-01
               :ARG1 (g / go
                        :ARG0 b)))
```

The *decode()* and *encode()* functions work with one PENMAN graph. The *load()* and *dump()* functions work with collections of graphs.

Users who want to work with trees would use *parse()* and *format()* instead:

```
>>> import penman
>>> t =  penman.parse('(w / want-01 :ARG0 (b / boy) :ARG1 (g / go :ARG0 b))')
>>> var, branches = t.node
>>> var
'w'
>>> len(branches)
3
>>> role, target = branches[2]
>>> role
':ARG1'
>>> print(penman.format(target))
(g / go
   :ARG0 b)
```

Contents

7.1 Module Constants

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

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

7.2 Classes

class penman.**Tree**
Alias of *penman.tree.Tree*.

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

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

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

7.3 Module Functions

7.3.1 Trees

penman.**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

```
>>> import penman
>>> penman.parse('(b / bark-01 :ARG0 (d / dog))')  # noqa
Tree(('b', [('/', 'bark-01'), (':ARG0', ('d', [('/', 'dog')]))]))
```

penman.**iterparse**(*lines*)

Yield trees parsed from *lines*.

Parameters **lines** – a string or open file with PENMAN-serialized graphs

Returns The *Tree* object described in *lines*.

Example

```
>>> import penman
>>> for t in penman.iterparse('(a / alpha) (b / beta)'):
...     print(repr(t))
...
Tree(('a', [('/', 'alpha')]))
Tree(('b', [('/', 'beta')]))
```

penman.**format**(*tree*, *indent=-1*, *compact=False*)

Format *tree* into a PENMAN string.

Parameters

- **tree** – a Tree object
- **indent** – how to indent formatted strings
- **compact** – if `True`, put initial attributes on the first line

Returns the PENMAN-serialized string of the Tree *t*

Example

```
>>> import penman
>>> print(penman.format(
...     ('b', [('/', 'bark-01'),
...            (':ARG0', ('d', [('/', 'dog')]))])))
(b / bark-01
   :ARG0 (d / dog))
```

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

Interpret a graph from the *Tree* *t*.

Alias of *penman.layout.interpret()*

7.3.2 Graphs

penman.**decode**(*s*, *model=None*)

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

Returns the Graph object described by *s*

Example

```
>>> import penman
>>> penman.decode('(b / bark-01 :ARG0 (d / dog))')
<Graph object (top=b) at ...>
```

penman.**iterdecode**(*lines*, *model=None*)

Yield graphs parsed from *lines*.

Parameters

- **lines** – a string or open file with PENMAN-serialized graphs
- **model** – the model used for interpreting the graph

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

Example

```
>>> import penman
>>> for g in penman.iterdecode('(a / alpha) (b / beta)'):
...     print(repr(g))
<Graph object (top=a) at ...>
<Graph object (top=b) at ...>
```

penman.**encode**(*g*, *top=None*, *model=None*, *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
- **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

```
>>> import penman
>>> from penman.graph import Graph
>>> penman.encode(Graph([('h', 'instance', 'hi')]))
'(h / hi)'
```

penman.**configure**(*g*, *top=None*, *model=None*)
Configure a tree from the *Graph* *g*.

Alias of *penman.layout.configure()*

7.3.3 Corpus Files

penman.**loads**(*string*, *model=None*)
Deserialize a list of PENMAN-encoded graphs from *string*.

Parameters

- **string** – a string containing graph data
- **model** – the model used for interpreting the graph

Returns a list of Graph objects

penman.**load**(*source*, *model=None*)
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

Returns a list of Graph objects

penman.**dumps**(*graphs*, *model=None*, *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
- **indent** – how to indent formatted strings
- **compact** – if `True`, put initial attributes on the first line

Returns the string of serialized graphs

penman.**dump**(*graphs*, *file*, *model=None*, *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
- **indent** – how to indent formatted strings
- **compact** – if `True`, put initial attributes on the first line

7.3.4 Triple Conjunctions

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

Example

```
>>> import penman
>>> for triple in penman.parse_triples('''
...         instance(b, bark) ^
...         ARG0(b, d) ^
...         instance(d, dog)'''):
...     print(triple)
('b', ':instance', 'bark')
('b', ':ARG0', 'd')
('d', ':instance', 'dog')
```

penman.**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

```
>>> import penman
>>> g = penman.decode('(b / bark-01 :ARG0 (d / dog))')
>>> print(penman.format_triples(g.triples))
instance(b, bark-01) ^
ARG0(b, d) ^
instance(d, dog)
```

7.4 Exceptions

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

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

7.5 Submodules

- *penman.codec* – Codec class for reading and writing PENMAN data
- *penman.constant* – For working with constant values
- *penman.epigraph* – Base classes for epigraphical markers
- *penman.exceptions* – Exception classes
- *penman.graph* – Classes for pure graphs
- *penman.layout* – Conversion between trees and graphs
- *penman.model* – Class for defining semantic models
- *penman.models* – Pre-defined models
- *penman.surface* – For working with surface alignments
- *penman.transform* – Graph and tree transformation functions
- *penman.tree* – Classes for trees

CHAPTER

EIGHT

PENMAN.CODEC

Serialization of PENMAN graphs.

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

An encoder/decoder for PENMAN-serialized graphs.

decode(*s*)

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

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

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

Example

```
>>> from penman.codec import PENMANCodec
>>> codec = PENMANCodec()
>>> codec.decode('(b / bark-01 :ARG0 (d / dog))')
<Graph object (top=b) at ...>
```

iterdecode(*lines*)

Yield graphs parsed from *lines*.

Parameters **lines** – a string or open file with PENMAN-serialized graphs

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

```
>>> from penman.codec import PENMANCodec
>>> codec = PENMANCodec()
>>> codec.parse('(b / bark-01 :ARG0 (d / dog))')  # noqa
Tree(('b', [('/', 'bark-01'), (':ARG0', ('d', [('/', 'dog')]))]))
```

iterparse(*lines*)

Yield trees parsed from *lines*.

Parameters **lines** – a string or open file with PENMAN-serialized graphs

Returns The *Tree* object described in *lines*.

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

encode(*g*, *top=None*, *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
- **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

```
>>> from penman.graph import Graph
>>> from penman.codec import PENMANCodec
>>> codec = PENMANCodec()
>>> codec.encode(Graph([('h', 'instance', 'hi')]))
'(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

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

CHAPTER

NINE

PENMAN.CONSTANT

Functions for working with constant values.

When a PENMAN string is parsed to a tree or a graph, constant values are left as strings or, if the value is missing, as `None`. Penman nevertheless recognizes four datatypes commonly used in PENMAN data: integers, floats, strings, and symbols. A fifth type, called a "null" value, is used when an attribute is missing its target, but aside from robustness measures it is not a supported datatype.

9.1 Enumerated Datatypes

`penman.constant.`**`SYMBOL = <Type.SYMBOL: 'Symbol'>`**
Symbol constants (e.g., `(... :polarity -)`)

`penman.constant.`**`STRING = <Type.STRING: 'String'>`**
String constants (e.g., `(... :op1 "Kim")`)

`penman.constant.`**`INTEGER = <Type.INTEGER: 'Integer'>`**
Integer constants (e.g., `(... :value 12)`)

`penman.constant.`**`FLOAT = <Type.FLOAT: 'Float'>`**
Float constants (e.g., `(... :value 1.2)`)

`penman.constant.`**`NULL = <Type.NULL: 'Null'>`**
Empty values (e.g., `(... :ARG1 )`)

9.2 Module Functions

`penman.constant.`**`type`**(*constant_string*)
Return the type of constant encoded by *constant_string*.

Examples

```
>>> from penman import constant
>>> constant.type('-')
<Type.SYMBOL: 'Symbol'>
>>> constant.type('"foo"')
<Type.STRING: 'String'>
>>> constant.type('1')
<Type.INTEGER: 'Integer'>
>>> constant.type('1.2')
```

(continues on next page)

(continued from previous page)

```
<Type.FLOAT: 'Float'>
>>> constant.type('')
<Type.NULL: 'Null'>
```

penman.constant.**evaluate**(*constant_string*)

Evaluate and return *constant_string*.

If *constant_string* is `None` or an empty symbol (`''`), this function returns `None`, while an empty string constant (`'""'`) returns an empty `str` object (`''`). Otherwise, symbols are returned unchanged while strings get quotes removed and escape sequences are unescaped. Note that this means it is impossible to recover the original type of strings and symbols once they have been evaluated. For integer and float constants, this function returns the equivalent Python `int` or `float` objects.

Examples

```
>>> from penman import constant
>>> constant.evaluate('-')
'-'
>>> constant.evaluate('"foo"')
'foo'
>>> constant.evaluate('1')
1
>>> constant.evaluate('1.2')
1.2
>>> constant.evaluate('') is None
True
```

penman.constant.**quote**(*constant*)

Return *constant* as a quoted string.

If *constant* is `None`, this function returns an empty string constant (`'""'`). All other types are cast to a string and quoted.

Examples

```
>>> from penman import constant
>>> constant.quote(None)
'""'
>>> constant.quote('')
'""'
>>> constant.quote('foo')
'"foo"'
>>> constant.quote('"foo"')
'"\\"foo\\""'
>>> constant.quote(1)
'"1"'
>>> constant.quote(1.5)
'"1.5"'
```

CHAPTER

TEN

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

ELEVEN

PENMAN.EXCEPTIONS

exception penman.exceptions.**PenmanError**
Base class for errors in the Penman package.

exception penman.exceptions.**ConstantError**
Bases: *penman.exceptions.PenmanError*

Raised when working with invalid constant values.

exception penman.exceptions.**GraphError**
Bases: *penman.exceptions.PenmanError*

Raised 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

TWELVE

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 *instances()*, *edges()* and *attributes()* methods return only those that are concept relations, relations between nodes, or relations 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

```
>>> from penman.graph import Graph
>>> Graph([('b', ':instance', 'bark-01'),
...        ('d', ':instance', 'dog'),
...        ('b', ':ARG0', 'd')])
<Graph object (top=b) at ...>
```

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.

instances()

Return instances (concept triples).

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 concept triples or those 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.**Instance**
Bases: *penman.graph.Triple*

A relation indicating the concept of a node.

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

THIRTEEN

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
```

13.1 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.

class penman.layout.**Pop**
Bases: *penman.layout.LayoutMarker*

Epigraph marker to indicate the end of a node context.

penman.layout.**POP = POP**
A singleton instance of *Pop*. Using the *POP* singleton can help reduce memory usage and processing time when working with many graphs, but it should **not** be checked for object identity, such as `if x is POP`, when working with multiple processes because each process gets its own instance. Instead, use a type check such as `isinstance(x, Pop)`.

13.2 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-01'),
...     ('ARG0', ('d', [
...       ('/', 'dog')]))]))
>>> g = layout.interpret(t)
>>> for triple in g.triples:
...     print(triple)
...
('b', ':instance', 'bark-01')
('b', ':ARG0', 'd')
('d', ':instance', 'dog')
```

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

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 role and returns some sortable criterion.

If the *attributes_first* argument is `True`, attribute branches are appear before any edges.

Instance branches (`/`) always appear before any other branches.

Example

```
>>> from penman import layout
>>> from penman.model import Model
>>> from penman.codec import PENMANCodec
>>> c = PENMANCodec()
>>> t = c.parse(
...    '(s / see-01'
...    '   :ARG1 (c / cat)'
...    '   :ARG0 (d / dog))')
>>> layout.rearrange(t, key=Model().canonical_order)
>>> print(c.format(t))
(s / see-01
   :ARG0 (d / dog)
   :ARG1 (c / cat))
```

13.3 Graph Functions

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

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

Returns The configured *Tree*.

Example

```
>>> from penman.graph import Graph
>>> from penman import layout
>>> g = Graph([('b', ':instance', 'bark-01'),
...            ('b', ':ARG0', 'd'),
...            ('d', ':instance', 'dog')])
>>> t = layout.configure(g)
>>> print(t)
Tree(
  ('b', [
    ('/', 'bark-01'),
    (':ARG0', ('d', [
      ('/', 'dog')]))]))
```

penman.layout.**reconfigure**(*g*, *top=None*, *model=None*, *key=None*)

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

If *key* is provided, triples are sorted according to the key.

13.4 Diagnostic Functions

penman.layout.**get_pushed_variable**(*g*, *triple*)

Return the variable pushed by *triple*, if any, otherwise `None`.

Example

```
>>> from penman import decode
>>> from penman.layout import get_pushed_variable
>>> g = decode('(a / alpha :ARG0 (b / beta))')
>>> get_pushed_variable(g, ('a', ':instance', 'alpha'))  # None
>>> get_pushed_variable(g, ('a', ':ARG0', 'b'))
'b'
```

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*.

penman.layout.**node_contexts**(*g*)

Return the list of node contexts corresponding to triples in *g*.

If a node context is unknown, the value `None` is substituted.

Example

```
>>> from penman import decode, layout
>>> g = decode('''
...    (a / alpha
...       :attr val
...       :ARG0 (b / beta :ARG0 (g / gamma))
...       :ARG0-of g)''')
>>> for ctx, trp in zip(layout.node_contexts(g), g.triples):
...     print(ctx, ':', trp)
...
a : ('a', ':instance', 'alpha')
a : ('a', ':attr', 'val')
a : ('a', ':ARG0', 'b')
b : ('b', ':instance', 'beta')
b : ('b', ':ARG0', 'g')
g : ('g', ':instance', 'gamma')
a : ('g', ':ARG0', 'a')
```

CHAPTER

FOURTEEN

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.

errors(*graph*)

Return a description of model errors detected in *graph*.

The description is a dictionary mapping a context to a list of errors. A context is a triple if the error is relevant for the triple, or `None` for general graph errors.

Example

```
>>> from penman.models.amr import model
>>> from penman.graph import Graph
>>> g = Graph([('a', ':instance', 'alpha'),
...            ('a', ':foo', 'bar'),
...            ('b', ':instance', 'beta')])
>>> for context, errors in model.errors(g).items():
...     print(context, errors)
...
```

(continues on next page)

(continued from previous page)

```
('a', ':foo', 'bar') ['invalid role']
('b', ':instance', 'beta') ['unreachable']
```

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_role_reifiable(*role*)
Return `True` if *role* 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*.

is_concept_dereifiable(*concept*)
Return `True` if *concept* can be dereified.

dereify(*instance_triple*, *source_triple*, *target_triple*)
Return the triple that dereifies the three argument triples.

If the target of *instance_triple* does not have a defined dereification, or if the roles of *source_triple* and *target_triple* do not match those for the dereification of the concept, a *ModelError* is raised. A ValueError is raised if *instance_triple* is not an instance triple or any triple does not have the same source variable as the others.

Parameters

- **instance_triple** – the triple containing the node's concept
- **source_triple** – the source triple from the node
- **target_triple** – the target triple from the node

Returns The triple that dereifies the three argument triples.

original_order(*role*)
Role sorting key that does not change the order.

alphanumeric_order(*role*)
Role sorting key for alphanumeric order.

canonical_order(*role*)
Role sorting key that finds a canonical order.

random_order(*role*)
Role sorting key that randomizes the order.

CHAPTER

FIFTEEN

PENMAN.MODELS

This sub-package contains specified instances of the *penman.model.Model* class.

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.

Roles


```
{
    ":ARG0":                {"type": "frame"},
    ":ARG1":                {"type": "frame"},
    ":ARG2":                {"type": "frame"},
    ":ARG3":                {"type": "frame"},
    ":ARG4":                {"type": "frame"},
    ":ARG5":                {"type": "frame"},
    ":accompanier":         {"type": "general"},
    ":age":                 {"type": "general"},
    ":beneficiary":         {"type": "general"},
    ":cause":               {"type": "general", "shortcut": true},
    ":concession":          {"type": "general"},
    ":condition":           {"type": "general"},
    ":consist-of":          {"type": "general"},
    ":cost":                {"type": "general", "shortcut": true},
    ":degree":              {"type": "general"},
    ":destination":         {"type": "general"},
    ":direction":           {"type": "general"},
    ":domain":              {"type": "general"},
    ":duration":            {"type": "general"},
    ":employed-by":         {"type": "general", "shortcut": true},
    ":example":             {"type": "general"},
    ":extent":              {"type": "general"},
    ":frequency":           {"type": "general"},
    ":instrument":          {"type": "general"},
    ":li":                  {"type": "general"},
    ":location":            {"type": "general"},
```


(continues on next page)

(continued from previous page)

```
    ":manner":               {"type": "general"},
    ":meaning":              {"type": "general", "shortcut": true},
    ":medium":               {"type": "general"},
    ":mod":                  {"type": "general"},
    ":mode":                 {"type": "general"},
    ":name":                 {"type": "general"},
    ":ord":                  {"type": "general"},
    ":part":                 {"type": "general"},
    ":path":                 {"type": "general"},
    ":polarity":             {"type": "general"},
    ":polite":               {"type": "general"},
    ":poss":                 {"type": "general"},
    ":purpose":              {"type": "general"},
    ":role":                 {"type": "general", "shortcut": true},
    ":source":               {"type": "general"},
    ":subevent":             {"type": "general"},
    ":subset":               {"type": "general", "shortcut": true},
    ":superset":             {"type": "general", "shortcut": true},
    ":time":                 {"type": "general"},
    ":topic":                {"type": "general"},
    ":value":                {"type": "general"},
    ":quant":                {"type": "quantity"},
    ":unit":                 {"type": "quantity"},
    ":scale":                {"type": "quantity"},
    ":day":                  {"type": "date"},
    ":month":                {"type": "date"},
    ":year":                 {"type": "date"},
    ":weekday":              {"type": "date"},
    ":timezone":             {"type": "date"},
    ":quarter":              {"type": "date"},
    ":dayperiod":            {"type": "date"},
    ":season":               {"type": "date"},
    ":year2":                {"type": "date"},
    ":decade":               {"type": "date"},
    ":century":              {"type": "date"},
    ":calendar":             {"type": "date"},
    ":era":                  {"type": "date"},
    ":op[0-9]+":             {"type": "op"},
    ":snt[0-9]+":            {"type": "snt"},
    ":prep-against":         {"type": "preposition"},
    ":prep-along-with":      {"type": "preposition"},
    ":prep-amid":            {"type": "preposition"},
    ":prep-among":           {"type": "preposition"},
    ":prep-as":              {"type": "preposition"},
    ":prep-at":              {"type": "preposition"},
    ":prep-by":              {"type": "preposition"},
    ":prep-for":             {"type": "preposition"},
    ":prep-from":            {"type": "preposition"},
    ":prep-in":              {"type": "preposition"},
    ":prep-in-addition-to":  {"type": "preposition"},
    ":prep-into":            {"type": "preposition"},
    ":prep-on":              {"type": "preposition"},
    ":prep-on-behalf-of":    {"type": "preposition"},
    ":prep-out-of":          {"type": "preposition"},
    ":prep-to":              {"type": "preposition"},
    ":prep-toward":          {"type": "preposition"},
    ":prep-under":           {"type": "preposition"},
```

(continues on next page)

(continued from previous page)

```
    ":prep-with":           {"type": "preposition"},
    ":prep-without":        {"type": "preposition"},
    ":conj-as-if":          {"type": "conjunction"}
}
```

Role Normalizations

```
{
    ":mod-of":    ":domain",
    ":domain-of": ":mod"
}
```

Reifications

```
[
    [":accompanier", "accompany-01",         ":ARG0", ":ARG1"],
    [":age",         "age-01",               ":ARG1", ":ARG2"],
    [":beneficiary", "benefit-01",           ":ARG0", ":ARG1"],
    [":beneficiary", "receive-01",           ":ARG2", ":ARG0"],
    [":cause",       "cause-01",             ":ARG1", ":ARG0"],
    [":concession",  "have-concession-91",   ":ARG1", ":ARG2"],
    [":condition",   "have-condition-91",    ":ARG1", ":ARG2"],
    [":cost",        "cost-01",              ":ARG1", ":ARG2"],
    [":degree",      "have-degree-92",       ":ARG1", ":ARG2"],
    [":destination", "be-destined-for-91",   ":ARG1", ":ARG2"],
    [":duration",    "last-01",              ":ARG1", ":ARG2"],
    [":employed-by", "have-org-role-91",     ":ARG0", ":ARG1"],
    [":example",     "exemplify-01",         ":ARG0", ":ARG1"],
    [":extent",      "have-extent-91",       ":ARG1", ":ARG2"],
    [":frequency",   "have-frequency-91",    ":ARG1", ":ARG2"],
    [":instrument",  "have-instrument-91",   ":ARG1", ":ARG2"],
    [":li",          "have-li-91",           ":ARG1", ":ARG2"],
    [":location",    "be-located-at-91",     ":ARG1", ":ARG2"],
    [":manner",      "have-manner-91",       ":ARG1", ":ARG2"],
    [":meaning",     "mean-01",              ":ARG1", ":ARG2"],
    [":mod",         "have-mod-91",          ":ARG1", ":ARG2"],
    [":name",        "have-name-91",         ":ARG1", ":ARG2"],
    [":ord",         "have-ord-91",          ":ARG1", ":ARG2"],
    [":part",        "have-part-91",         ":ARG1", ":ARG2"],
    [":polarity",    "have-polarity-91",     ":ARG1", ":ARG2"],
    [":poss",        "own-01",               ":ARG0", ":ARG1"],
    [":poss",        "have-03",              ":ARG0", ":ARG1"],
    [":purpose",     "have-purpose-91",      ":ARG1", ":ARG2"],
    [":role",        "have-org-role-91",     ":ARG0", ":ARG2"],
    [":source",      "be-from-91",           ":ARG1", ":ARG2"],
    [":subevent",    "have-subevent-91",     ":ARG1", ":ARG2"],
    [":subset",      "include-91",           ":ARG2", ":ARG1"],
    [":superset",    "include-91",           ":ARG1", ":ARG2"],
    [":time",        "be-temporally-at-91",  ":ARG1", ":ARG2"],
    [":topic",       "concern-02",           ":ARG0", ":ARG1"],
    [":value",       "have-value-91",        ":ARG1", ":ARG2"],
    [":quant",       "have-quant-91",        ":ARG1", ":ARG2"]
]
```

15.1.2 penman.models.noop

No-op semantic model definition.

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

Bases: *penman.model.Model*

A no-operation model that mostly leaves things alone.

This model is like the default *Model* except that *NoOpModel.deinvert()* always returns the original triple, even if it was inverted.

deinvert(*triple*)

Return *triple* (does not deinvert).

penman.models.noop.**model**

An instance of the *NoOpModel* class.

CHAPTER

SIXTEEN

PENMAN.SURFACE

Surface strings, tokens, and alignments.

16.1 Epigraphical Markers

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

Bases: *penman.epigraph.Epidatum*

classmethod from_string(*s*)

Instantiate the alignment marker from its string *s*.

Examples

```
>>> from penman import surface
>>> surface.Alignment.from_string('1')
Alignment((1,))
>>> surface.RoleAlignment.from_string('e.2,3')
RoleAlignment((2, 3), prefix='e.')
```

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.

Example

```
>>> from penman import decode
>>> from penman import surface
>>> g = decode(
...    '(c / chase-01~4'
...    '   :ARG0~5 (d / dog~7)'
...    '   :ARG0~3 (c / cat~2))')
>>> surface.alignments(g)
{('c', ':instance', 'chase-01'): Alignment((4,)),
 ('d', ':instance', 'dog'): Alignment((7,)),
 ('c', ':instance', 'cat'): Alignment((2,))}
```

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.

Example

```
>>> from penman import decode
>>> from penman import surface
>>> g = decode(
...    '(c / chase-01~4'
...    '   :ARG0~5 (d / dog~7)'
...    '   :ARG0~3 (c / cat~2))')
>>> surface.role_alignments(g)
{('c', ':ARG0', 'd'): RoleAlignment((5,)),
 ('c', ':ARG0', 'c'): RoleAlignment((3,))}
```

CHAPTER

SEVENTEEN

PENMAN.TRANSFORM

Tree and graph transformations.

See also:

The transformation functions in this module alter the content of the graph. Other functions may change the shape or form of the graph without altering its content, such as:

- *penman.layout.rearrange()*
- *penman.layout.reconfigure()*
- *penman.tree.Tree.reset_variables()*

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
>>> 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.**dereify_edges**(*g*, *model*)

Dereify edges in *g* that have reifications in *model*.

Parameters **g** – a *Graph* object

Returns A new *Graph* object with dereified edges.

Example

```
>>> from penman.codec import PENMANCodec
>>> from penman.models.amr import model
>>> from penman.transform import dereify_edges
>>> codec = PENMANCodec(model=model)
>>> g = codec.decode(
...    '(c / chapter'
...    '   :ARG1-of (_ / have-mod-91'
...    '               :ARG2 7))')
>>> g = dereify_edges(g, model)
>>> print(codec.encode(g))
(c / chapter
   :mod 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)
>>> 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.

reset_variables(*fmt='{prefix}{j}'*)
Recreate node variables formatted using *fmt*.

The *fmt* string can be formatted with the following values:

- `prefix`: first alphabetic character in the node's concept
- `i`: 0-based index of the current occurrence of the prefix
- `j`: 1-based index starting from the second occurrence

walk()
Iterate over branches in the tree.

This function yields pairs of (*path*, *branch*) where each *path* is a tuple of 0-based indices of branches to get to *branch*. For example, the path (2, 0) is the concept branch *('/', 'bark-01')* in the tree for the following PENMAN string, traversing first to the third (index 2) branch of the top node, then to the first (index 0) branch of that node:

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

The (*path*, *branch*) pairs are yielded in depth-first order of the tree traversal.

penman.tree.**is_atomic**(*x*)
Return `True` if *x* is a valid atomic value.

Examples

```
>>> from penman.tree import is_atomic
>>> is_atomic('a')
True
>>> is_atomic(None)
True
>>> is_atomic(3.14)
True
>>> is_atomic(('a', [('/', 'alpha')]))
False
```

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

p

INDEX

L

M

N

O

P

Q

R

S

T

V

W