Encyclopedia Documentation

Release 0.44

Scott Howard James

Feb 24, 2023

A Brief Tour

1 Overview **1**
1.1 What Is It? . . . 1
1.2 But What Is It Really . . . 1
1.3 Say that Differently . . . 1

2 Ok, But What is it Good For? **3**
2.1 Relation . . . 3
2.2 Forest . . . 6
2.3 Dictionary . . . 7
2.4 Record . . . 7

3 Encyclopedia Operations **9**
3.1 Signed Encyclopedia . . . 10
3.2 Indexed Encyclopedia . . . 10

4 Past, Present and Future **13**
4.1 Past . . . 13
4.2 Present . . . 13
4.3 Future . . . 14

5 Abstract Encyclopedias **15**
5.1 Unindexed Encyclopedia . . . 15
5.2 Indexed Encyclopedia . . . 15
5.3 Signed Encyclopedia . . . 16

6 Concrete Encyclopedias **17**
6.1 Dictionary . . . 17
6.2 EAV . . . 17
6.3 Record . . . 18
6.4 Relation . . . 19
6.5 Forest . . . 20
6.6 Arboretum . . . 21
6.7 XML . . . 21
6.8 KML . . . 22

7 Index **23**

Index **25**

CHAPTER 1

Overview

1.1 What Is It?

An Encyclopedia is an abstract container intended for smallish data sets which provides many of the benefits of a relational (and non-relational) database, but with lower syntactic friction. In particular, an Encyclopedia uses arithmetic expressions typical of core storage mechanisms (e.g. lists, dictionaries) to perform common dataset operations such as merging and subsetting. Encyclopedia supports functional composition, enabling modifications to entire Encyclopedias in a single statement, similar to data frame vectorization.

1.2 But What Is It Really . . .

An **Encyclopedia** is a mapping class supporting the following additional features:

set operations Encyclopedias may be created and combined using standard arithmetic operators

functional composition Encyclopedia contents may be modified in their entirety by functions, as well as other Encyclopedias

and the following optional features:

multi-valued assignments keys may be *assigned* single values or *tagged* with multiple values

inversion Encyclopedias may swap their key-value pairs

1.3 Say that Differently

In math-speak, a collection of Encyclopedias is a similar to a mathematical ring of relations where:

- keys serve as the *domain*
- values serve as the *range*
- ring *addition* is performed by set union on these relations

- *additive inverse* is performed by set difference on these relations
- *multiplication* is performed by functional composition

So to say this yet another way, an Encyclopedia is a collection of key-value pairs which may be combined with other Encyclopedias using set operations and functional operators.

CHAPTER 2

Ok, But What is it Good For?

Data analysts have an embarrassment of riches when it comes to manipulating their data. We have multiple computational elements sitting on our laptop and have immediate access to nearly unlimited computational elements in The Cloud. We have various topologies to help access and store our data including: tabular (e.g. HDF5), relational (e.g. SQL) and non-relational (e.g. NoSQL). We have data-analyst-friendly languages (e.g. python, R) with increasingly sophisticated libraries (e.g. SciPy, tidyverse) to make it all fit together.

On the more modest end of the data deluge resides the localized data: the tabular CSV reports, the hierarchical XML data elements and the binary, human-friendly tag clouds. The Encyclopedia syntax is intended to be used in this realm, providing a common syntax for the "smaller" data elements, providing a bridge between the scripting container elements (e.g. lists, dictionaries) and the larger data ocean.

As an abstract class, Encyclopedia has limited value on its own. Two *concrete* Encyclopedias however, are Relations and Forests, which were the motivators for the creation of the Encyclopedia abstraction.

2.1 Relation

A Relation is an Indexed Encyclopedia which may be thought of as a multi-valued extension of a python dictionary or function.

In addition to the many-to-one behavior of a Dictionary, a Relation supports all four cardinalities:

M:1 (many-to-one) a function or dictionary

1:1 (one-to-one) an isomorphism or unique alias

1:M (one-to-many) a partition

M:M (many-to-many) a general relation

A Relation instance is restricted to one of the cardinalities upon object instantiation. Relations are invertible[1], providing direct mappings from values back to keys. The following operators are supported:

[1] all relations are "invertible" in the sense that domain/range may be swapped; however, relations composed with their inverse will only create Unity properly when the cardinality is 1:M or 1:1

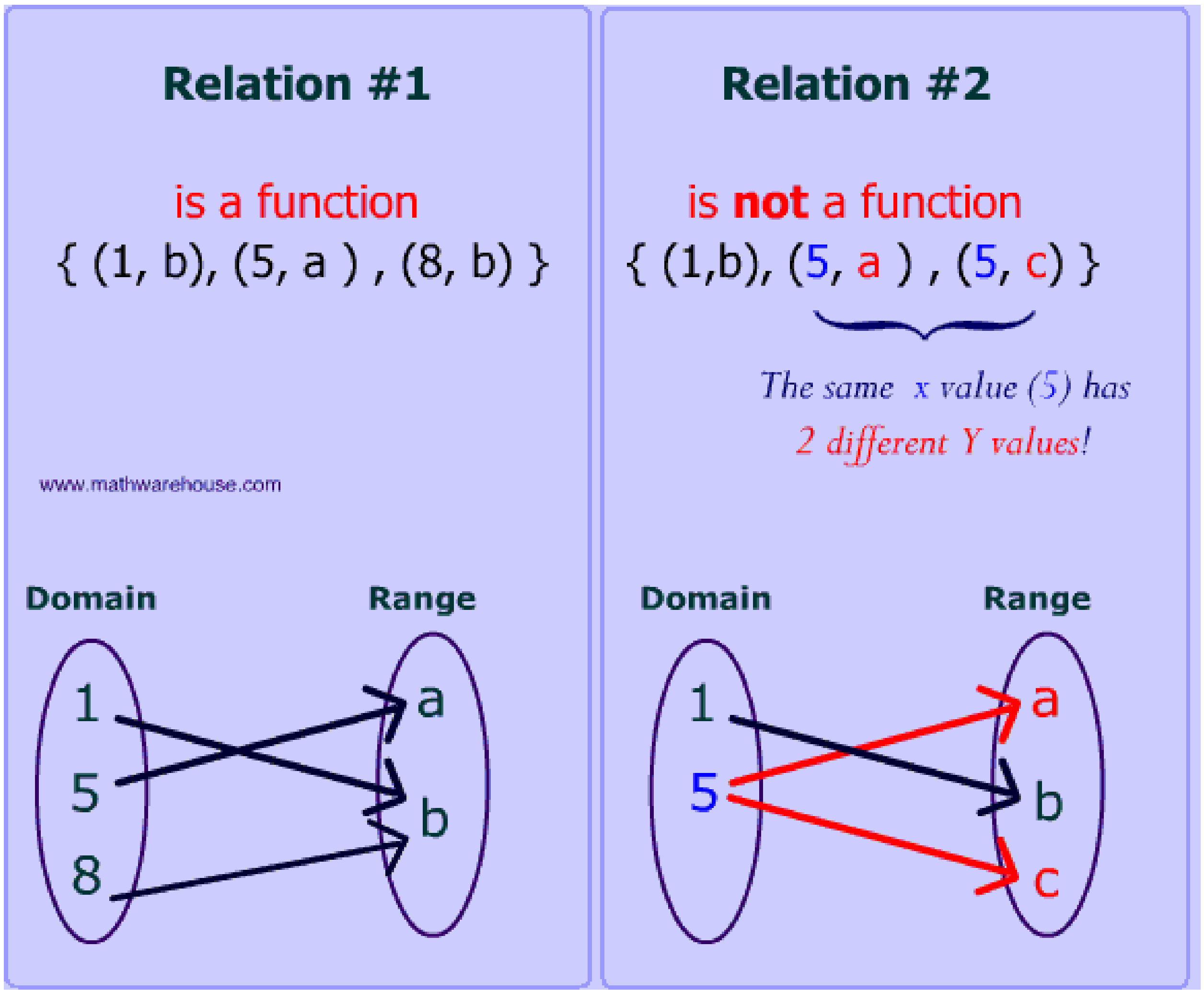

Fig. 1: Function vs. Relation

Notation	Meaning
R[x] = y	either overwrite or append to *x* values depending on cardinality of the Relation (Note: M:1 and 1:1 overwrite, the other two append)
del R[x]	remove *x* from domain of *E* and all associated values for *x*
R1 + R2	similar to `{**R1, **R2}` for python dictionaries, but with associated cardinality constraints
R1 - R2	remove any *R2*.keys that lie within *R1*.keys and the associated values
f * R	apply f to each element of R
R1 * R2	apply R1 to each element of R2
~R	reverse domain and range of R

Using notation similar to a python dictionary comprehension (but to be clear not *actually* valid python), a functional composition might be expressed as:

```
{R[f(x)]:f(R[x]) for x in R}
```

A relational composition as:

```
{R1[x]:R1[R2[x]) for x in R2}
```

And an inversion as:

```
{R[x]:x for x in R}
```

As an example of the use of a relation, suppose we need to map qualitative weather conditions to dates:

```
weather = Relation()
weather['2011-7-23']='high-wind'
weather['2011-7-24']='low-rain'
weather['2011-7-25']='low-rain'
weather['2011-7-25']='low-wind'
```

Note that in the last statement the assignment operator performs an append not an overwrite. So:

```
weather['2014-7-25']
```

produces a *set* of values:

```
{'low-rain','low-wind'}
```

Relation also provides an inverse:

```
(~weather)['low-rain']
```

also producing a set of values:

```
{'2014-7-25','2014-7-24'}
```

See the paper from SciPy 2015 for further exposition on Relation.

2.2 Forest

Forests are Unindexed Encyclopedias formed from collections of trees.

Syntactically a tree, in our parlance, will grow "upwards"; thus the greater heights of a tree will be closer to the "leaves". Each *node* in a tree connects upwards to a collection of distinct nodes; conversely each node has at most a single, directly-connecting lower node. Forests may be combined with other Forests using set operations (*horizontal combination*), and be grown on top of other Forests using composition (*vertical combination*).

Sub-branches of Forests are obtained through the bracket "get" notation:

```
F[x]
```

Note that the *keys* used in this bracket notation are different than *nodes*. In particular, *nodes* within a Forest are unique; however, *keys* may reference multiple nodes. Therefore, there is a many-to-one relationship between keys and nodes; thus, the "get" returns *all* sub-branches in F with a root node *keyed* by *x*.

To construct new branches, Forests use the "set" bracket notation. The bracket notation of Forests allows for several nodes to be *referenced* by a single key, specifically:

```
F[x] = y
```

means: create a new node, keyed by *y*, *for every* node that is keyed by *x*.

Forests form the topological foundation of many common hierarchical document formats e.g. XML, JSON, YAML etc… Non-unique keys enable us to include repeated substructures. For instance, the get notation in another context, namely when *y* is another forest:

```
F1[x] = F2
```

grafts the F2 Forest to *all* occurrences of *x* within F1. An example of a related operation is a YAML alias. This grafting can also be performed using composition notation:

```
F1 * F2
```

which means: create a new Forest such when F1 and F2 share a key *x*, the branches of F2[x] are grafted onto F1 at *x*. An example of a related operation is when a library of sub-documents are instanced onto a document when ready for final document production. The operations for a Forest are as follows:

Notation	Meaning
F[x] = y	connect new nodes keyed by *y* to nodes keyed by *x*
F[x]	a Forest consisting all nodes reachable from *x*
F[x] = F2	graft *F2* to *F1* at *x*
del F[x]	prune branches for all nodes keyed by *x*
F.keys()	return all node *keys* within Forest
F.values()	all *nodes* within Forest
F.canopy()	union of all leaf nodes in Forest
F.root(x)	return node(s) of Tree root containing *x*
F1 + F2	combine two Forests such that common Trees within both Forests will only appear once (*union*)
F1 - F2	remove Trees contained in *F2* from *F1* (*difference*)
F1 * F2	for each *x* key common to *F1* and *F2*: graft *F2* onto *F1* at *x*.
f * F	apply f to each node of F

An extension of a Forest is an Arboretum: a Forest with inheritable node attributes. Attributes are assigned using the second position in the bracket assignment, namely:

```
F[x, attribute] = value
```

This assigns the key-value pair *(attribute, value)* directly to *x* as well as implicitly to the nodes above *x*. Retrieving attributes is dynamic:

```
F[x, attribute]
```

meaning, the tree is searched for an attribute starting at the node and descending down the tree until a parent is found with the assignment. As a motivating example, suppose we had a hierarchical document:

```
F['Document'] = 'Section 1'
F['Section 1'] = 'Section 1.1'
```

Assigning the font

```
F['Section 1', 'font'] = 'Helvetica'
```

will affect *Section 1* and *Section 1.1* but will not affect the overall document. A new section created at the *Document* level

```
F['Document'] = 'Section 2'
```

will be unaffected by the font assignment but further subsections below *Section 1.1*

```
F['Section 1.1'] = 'Section 1.1.1'
```

will have their default font set.

2.3 Dictionary

Another example of an Encyclopedia is simply a python dictionary which has been Encyclopedia-ified. This new dictionary will behave much like its derived *dict* but will also support arithmetic set operations and composition. As an example of the composition feature, if:

```
fruit = Dictionary({'apple':'red', 'blueberry':'blue'})
colors = Dictionary({'red':'FF0000', 'blue':'0000FF', 'green':'00FF00'})
```

then

```
fruit * colors == Dictionary({'apple': 'FF0000', 'blueberry': '0000FF'})
```

2.4 Record

Building on the Dictionary, a Record is factory for Dictionaries providing other features including:

- restricted keys
- automatic type conversions
- optional defaults

As an example, we can define a factory for recording personal characteristics:

```
from encyclopedia import Record
characteristics = Record({
    'name':str,
    'age':int})
```

To create the individual Dictionaries we use the `Record.instance` function:

```
Dog = characteristics.instance # make it more class-like
fido = Dog()
fido['age']='2'
assert fido['age']==2
```

Note that the `age` was converted to an integer by the *function* `int`. We can put in any functions we like into the Record defintion and even auto-populate:

```
def no_name(x=None):
    return 'UNKNOWN' if x is None else str(x)

Named = Record({
    'name':no_name,
    'age':int},
    autopopulate=True).instance

someone = Named()
assert someone['name']=='UNKNOWN'
```

Note also that Record will complain (with a friendly Exception) is one were to attempt to set keys other than provided in the factory:

```
someone['address'] # not valid
```

CHAPTER 3

Encyclopedia Operations

It may be illustrative (at least for those of us who *like* looking at summary tables) to now show an overview of operations for an Encyclopedia:

Operation	Description
E[x] = y	tag *x* with *y*
del E[x]	remove *x* from domain of *E* … also known as burglary. Yep, that was your obligatory Monty Python reference
E1 + E2	an encyclopedia created by combining elements of *E2* and *E1* (*union*)
E1 += E2	add copy of *E2* to *E1*
E1 - E2	an encyclopedia created by removing keys of *E2* from *E1* (*difference*)
E1 & E2	an encyclopedia with keys common to both *E1* and *E2* (intersection)
f * E	apply *f*[2] to all elements of *E* returning another encyclopedia. (*functional composition*)
E1 * E2	apply *E1*[3] to elements of *E2* producing another encyclopedia (*entity composition*)

Certain implementations of Encyclopedia may be multi-valued, meaning that, assignment:

```
E[x] = y
```

may not *overwrite* the key's value, but instead *append* to the key value or *tag* the key. Similarly, retrieval:

```
E[x]
```

may produce a set (or list) of values corresponding to the key.

For the math-letes, note that encyclopedia addition is inherently commutative:

```
E1+E2 == E2+E1
```

and associative:

[2] when *f* is a scalar, assume function is multiplicative

[3] what encyclopedia composition actually *means* will depend on the specific encyclopedia implementation, but the *intention* of composition is to act element-wise, that is independently of other elements in the encyclopedia

```
E1+(E2+E3) == (E1+E2)+E3
```

due to the nature of element-wise set operations. Composition, however, is *not* necessarily commutative:

```
E1*E2 ?= E2*E1
```

but it is distributive[4]:

```
E1*(E1+E3) == E1*(E2+E2)
```

as functions act element-wise on the keys.

3.1 Signed Encyclopedia

An Encyclopedia is not a proper ring without the existence of a negative signed Encyclopedia:

```
-E
```

Note that we specifically refer to the unary operation and not the binary set difference. Note too that this is a little conceptually unusual, as a negative encyclopedia behaves a bit like antimatter, able to negate a collection of key-values, but not necessarily to serve as a meaningful mapping in our eminently practical universe. If the unary negative sign is supported by a derived Encyclopedia, the class will be known as a **Signed Encyclopedia**, and the following features will also be supported:

Identity	Field
Null[x]	existence of Null operator producing None or Error depending on implementation
abs(E)	invert “sign” of Encyclopedia
E + abs(E)	retain only positive (“real”) components of the Encyclopedia
abs(E1-E2) + abs(E2-E1)	*symmetric difference*
(Null + E)[x] == E[x]	*additive identity*
(E - E)[x] == Null[x]	*additive inverse*

One important distinction between a Signed Encyclopedia and an Unsigned Encyclopedia is the implementation of the intersection. For an Unsigned Encyclopedia, we may simply remove the elements of `E1` which are not in `E2`:

```
E1&E2 == E1-(E1-E2)
```

For a Signed Encyclopedia however, this won’t work as `-E1` is another Encyclopedia:

```
E1-(E1-E2) == E1-E1+E2 == E2
```

Instead we must use the Signed Encyclopedia’s *abs* operator to remove the negative elements first:

```
E1&E2 == E1-abs(E1-E2)
```

3.2 Indexed Encyclopedia

When a multiplicative inverse:

[4] what encyclopedia composition actually *means* will depend on the specific encyclopedia implementation, but the *intention* of composition is to act element-wise, that is independently of other elements in the encyclopedia

```
~E
```

is available, the Encyclopedia is a field where:

```
~E*E == ~E*E == Identity
```

that is,

```
(~E*E)[x]==x
```

Finally, an *Indexed* Encyclopedia supports inversion, including the following operators and identities:

Notation	Meaning
Unity[x] == x	existence of *unity*
~E	swap domain and range of *E* (*multiplicative inverse*)
(E*~E)[x] == (~E*E)[x] == x	Encyclopedia composed with its inverse produces Unity
(Unity * E)[x] == E[x]	Unity composed with an Encyclopedia produces that Encyclopedia

CHAPTER 4

Past, Present and Future

4.1 Past

As with many abstract types, the concept of Encyclopedia did not emerge from the void ready to be forward instantiated, but rather resulted from the backwards abstraction of specific, concrete implementations (not surprisingly to anyone following along at this point): Relations and Forests. These classes, in turn, were created to scratch particular itches:

- **Relation**: generalize the notion of a python dictionary to allow for many-to-many relations and provide other conveniences such as invertibility
- **Forest**: provide a tree syntax using standard mathematical notation which can then be used to construct various hierarchical data structures

Syntax may not be everything, but it helps. A lot. As many data analysts have found, being able to express something conveniently may determine whether the analysis gets done *at all*. Indeed, much of the power of scripting languages, including python, is the ability to express more complex structures, since the foundational structures (e.g. lists, sets, dictionaries) are so easy to describe.

Addressing Forests specifically, there are a number of different hierarchical structures (e.g. YAML, XML, JSON) which are each essentially trees, topologically, but are supported by different packages and syntaxes. Moreover, with regard to content generation, they sometimes lack the syntax for easily building more complex trees from simpler ones, such as, mentioned above, combining two trees either as a simple union or recursively, with one tree nested inside the other.

4.2 Present

The Encyclopedia specification, and implementations for:

- Relation
- Forest
- Arboretum

- Dictionary
- Record

as well as an:

- Encyclopedic wrapper for XML

Can be obtained at

- Github: https://github.com/scott-howard-james/encyclopedia
- PyPi: https://pypi.python.org/pypi/encyclopedia/ (alternatively, just `pip encyclopedia`)

Note that Encyclopedia has no dependencies outside of the standard python distribution.

4.3 Future

In the near-future, wrappers will be included for YAML and JSON. Additionally, support for other graph types will be added.

CHAPTER 5

Abstract Encyclopedias

5.1 Unindexed Encyclopedia

`class encyclopedia.`**`Unindexed`**`(`*mapping=None, frozen=False*`)`

Bases: `collections.abc.MutableMapping`

An *Unindexed Encyclopedia* extends a *MutableMapping* with the following features:

- **composition**: Encyclopedia contents may be altered by functions or Encyclopedias
- **set operation**: Encyclopedias may be combined using union, difference and intersection
- **mutability**: ability to “freeze” and “melt” an object

`__add__``(`*other*`)`

create union with this encyclopedia and another

`__mul__``(`*other*`)`

compose encyclopedia with another object

`__sub__``(`*other*`)`

perform set difference of this encyclopedia and other

`copy``()`

perform a deep copy

`freeze``()`

make encyclopedia immutable

`melt``()`

make encyclopedia mutable

5.2 Indexed Encyclopedia

`class encyclopedia.`**`Indexed`**`(`*mapping=None, frozen=False*`)`

Bases: `encyclopedia.templates.Unindexed`

An **Indexed** Encyclopedia may be inverted such that its values map to their keys

`__invert__()`
invert the Encyclopedia

5.3 Signed Encyclopedia

class `encyclopedia.`**`Signed`**(*mapping=None*, *frozen=False*)

Bases: `encyclopedia.templates.Indexed`

A *Signed Encyclopedia* may contain negative elements, that is, elements which "cancel" similarly keyed elements

`__abs__`()
remove negative elements for this encyclopedia

`__and__`(*other*)
intersect using signed logic

`__neg__`()
negate this encyclopedia

CHAPTER 6

Concrete Encyclopedias

6.1 Dictionary

class `encyclopedia.`**`Dictionary`**(*mapping=None*, *frozen=False*)

Bases: `dict`, `encyclopedia.templates.Unindexed`

A simple instantiation adding Encyclopedic features to the python dictionary

6.2 EAV

class `encyclopedia.`**`EAV`**(*data=None*, *fmt: str = None*, *fields=('entity', 'attribute', 'value')*, *vcast=None*, *acast=<class 'str'>*, *ecast=<class 'str'>*, *defaults=None*, *vcasts=None*)

Bases: `dict`, `encyclopedia.templates.Unindexed`

Container for storing smallish EAV "triples" (Entity-Attribute-Value). - intent of class is to provide dictionary-like access rather than data analysis functionality - internally, EAV is stored as a dictionary (key:E) of dictionaries (key:A,value:V) - class supports encyclopedic operations e.g. subtraction (difference) and addition (union)

set-ting:

eav[entity, attribute] = value eav[[entity1, …], attribute] = value eav[[entity1, …], attribute] = [value1, …] # len(entities) must equal len(values) eav[:, attribute] = value # assign all entities same value for attribute

get-ting:

eav[entity, attribute] # value for a specific attribute eav[entity] # dictionary of elements referenced by entity

get-ting producing new EAV:

eav[:, attribute] # new EAV with all entities and but only one attribute eav[:, [attribute1, attribute2]] # new EAV with all entities and but only specified attributes eav[entity,:] # new EAV with only one entity eav[[entity1, …],:] # new EAV with only specified entities

Unsupported at this time:

> eav[entity, :] = value # ERROR eav[entity, [attribute1, . . .]] = [value1, . . .] # ERROR

`__delitem__`(*thing*)
: Delete self[key].

`__getitem__`(*thing*)
: x.__getitem__(y) <==> x[y]

`__init__`(*data=None*, *fmt: str = None*, *fields=('entity', 'attribute', 'value')*, *vcast=None*, *acast=<class 'str'>*, *ecast=<class 'str'>*, *defaults=None*, *vcasts=None*)

- **fmt (type may be specified using one of the following *strings* . . .)**
  - dict: dictionary of dictionaries (auto-detected)
  - triple: list of EAV dictionaries/tuples (defaulted)
  - column: list of records with field names as first row and entities on first column (must force this option)
- vcast: value cast e.g. int
- acast: attribute cast e.g. str
- ecast: entity cast e.g. str
- defaults: dictionary of defaults for specific attributes
- vcasts: dictionary of casting for specific attributes

`__setitem__`(*thing*, *value*)
: Set self[key] to value.

`__str__`()
: Return str(self).

`__weakref__`
: list of weak references to the object (if defined)

`attributes`(*entities=None*)
: computationally determine which attributes are used for specified entities

`compose`(*other*, *entities=None*)
: perform functional or Encyclopedic composition

`copy`()
: deep copy of EAV. Preserves casting and defaults.

`copy_style`()
: create empty EAV preserving casting and defaults.

`rename`(*renames*, *entities=None*)
: rename attributes (. . . not the entities)

`subtract`(*other*)
: perform set difference

6.3 Record

`class` `encyclopedia.`**`Record`**(*mapping=None*, *autopopulate=False*, *restrict=True*)
: Bases: `dict`, `encyclopedia.templates.Unindexed`

A factory creating Encyclopedia-ified Dictionaries with:

- restricted keys
- automatic type conversions
- optional defaults

`instance``()`
  create an individual (Record) instance

6.4 Relation

`class` `encyclopedia.`**`Relation`**`(`*init=None*, *cardinality='M:M'*, *ordered=False*, *frozen=False*`)`
  Bases: `encyclopedia.templates.Indexed`

  General purpose, discrete relation container for all four mapping cardinalities:

- 1:1 (*Isomorphism*)
- 1:M (*Immersion*)
- M:1 (*Function* e.g. Python Dictionary)
- M:M (*General Relation*)

  Inversion, for all cardinalities, is provided (at the cost of doubled storage)

  `exception Error`
    Bases: `encyclopedia.templates.Error`

    label Relation exceptions

  `__init__``(`*init=None*, *cardinality='M:M'*, *ordered=False*, *frozen=False*`)`
    create a new Relation using a variety of different inputs

  `__invert__``()`
    reverse the domain and range

    Note: Relation inversion uses references instead of copies

  `__len__``()`
    number of keys (domain)

  `__setitem__``(`*domain*, *target*`)`

  - add key-value pairs for 1:M and M:M cardinalities
  - overwrite key values for 1:1 and M:1 cardinalities

  `__str__``()`
    display the forward and inverted mappings

  `compose``(`*other*`)`
    compose relation with another relation or a function

  `subtract``(`*other*`)`
    implement set difference

  `unfrozen``()`
    function decorator to check if object is unfrozen

  `update``(`*other*`)`
    update a Relation with another Relation

values()
  the range of the mapping

6.5 Forest

class encyclopedia.**Forest**(*offset: int = 0*, *parent=None*)
  Bases: `encyclopedia.templates.Unindexed`

  An Encyclopedia of trees <https://en.wikipedia.org/wiki/Tree_(graph_theory)>. There is no specific "Tree" class; instead, a Tree is Forest with a single, connected graph. This is purposeful, as adding two Trees creates a Forest, not another Tree.

  Terminology for the Forest class:

  - *Tree*: single, connected (tree) graph
  - *Forest*: a (possibly empty) collection of Trees, supporting encyclopedic operations
  - *Node*: a single node in a graph
  - *Twig*: a connected pair of nodes
  - *Sprout*: a new, connected node

  class Node(*alias*, *id*)
    Bases: `object`

    store the non-unique alias and the unique ID for a Node

    identified(*identified: bool*, *off: int = 0*)
      return either an offset Node ID or simply the alias

  above(*alias*, *aliased=False*)
    generate nodes above aliased node(s)

  aliased(*alias=None*)
    generate set of nodes referenced by an alias

  below(*alias*, *aliased=False*)
    generate parent below aliased node(ss)

  branches(*alias*)
    return branches (e.g. Trees) above the aliased node(s)

  climb(*alias=None*, *level: bool = False*, *twig: bool = False*, *identified: bool = True*, *offset: int = 0*)
    generate tree of nodes reacheable from alias

  compose(*other*)
    composition depends on context of "other":

    - scaled by integers
    - acted upon by functions
    - grafted with Forests

  cutting(*alias=None*, *identified: bool = True*, *offset: int = 0*, *morph: function = <function Unindexed.identity>*)
    subforest rooted at alias

  graft(*forest*, *alias=None*, *identified: bool = True*)
    glue forest onto another forest at aliased node(s)

height()
: the maximum height of all nodes in Forest (in all trees)

leaves(*alias=None*)
: leaves reachable from alias

limb(*alias*, *offset: int = 0*)
: the tree (single line of nodes) below aliased node(s)

prune()
: eliminate duplicate trees in a forest

root(*alias=None*)
: tree root of aliased node(s)

sorted(*alias=None*, *level: bool = False*, *twig: bool = False*, *offset: int = 0*, *identified: bool = True*, *aliased: bool = False*)
: topologically sorted node *list* (not iterator)

sprout(*alias*)
: make a unique node identifier (referenced by an alias)

subtract(*other*)
: perform set difference when "other" is a Forest; otherwise, delete element

  Note: Encyclopedia mixin

unfrozen()
: function decorator to check if object is unfrozen

update(*other*)
: perform set union when "other" is a Forest; otherwise, add a new tree

  Note: MutableMapping mixin

values()
: implicit by __getitem__ (mixin) but implemented for performance reasons

6.6 Arboretum

class `encyclopedia.`**Arboretum**(*offset: int = 0*, *parent=None*)
: Bases: `encyclopedia.forest.Forest`

  A Forest with inhertiable attributes. Note: as attributes are assigned using setitem syntax, tuples cannot be used as node aliases

6.7 XML

class `encyclopedia.`**XML**
: Forest syntax combined with an elementTree XML implementation. To avoid internal confusion, inherits from neither elementTree nor Arboretum directly, thus the tree(forest) is stored *in parallel* to the elementTree structure

copy()
: perform a deep copy

unique(*name*)
: create a unique element name

write (*filename=None*, *doctype=None*)
write the XML, adding tabs for pretty-printing

6.8 KML

class encyclopedia.**KML**
An encyclopedia-ified version of the KML (Keyhole Markup Language) GIS format. In addtion to encyclopedia operations, KML supports the following features:

- a draw function for ease in creating a KML geometries
- records for managing KML styles and coordinates

Coordinate
alias of `encyclopedia.record.Record.instance.<locals>.Internal`

Style
alias of `encyclopedia.record.Record.instance.<locals>.Internal`

static coordinated (*record*, *show_description=None*, *show_data=None*, *altitude_in_feet=True*)
create a KML coordinate

draw (*points*, *folder*, *style*, *geometry='LineString'*, *extrude=False*, *visibility=True*, *tesselate=None*, *altitude='absolute'*)
create a KML geometry in a specified folder using:

- collection of KML drawing parameters
- set of points

points are provided as an iterated list of dictionaries with (at least) the following fields:

- uid (Unique Identifier)
- id (KML element label)
- lat
- lon
- alt (in feet)
- tick (UNIX time)

Points will be plotted in the order they are occur in the stream Meta data (e.g. id) will only be taken from the first point in the stream

if begin/end appears as fields:

- begin (UNIX time)
- end (UNIX time)

will create a time block. As with meta data, only the data on the first uid appearance is used

static styled (*record*)
create a KML style

stylize (*record*)
create a KML style element

write (*filename=None*, *doctype=None*, *mlns='http://www.opengis.net/kml/2.2'*, *gxmlns='http://www.google.com/kml/ext/2.2'*)
write KML to a file

CHAPTER 7

Index

- genindex

Index

Symbols

`__abs__()` (*encyclopedia.Signed method*), 16
`__add__()` (*encyclopedia.Unindexed method*), 15
`__and__()` (*encyclopedia.Signed method*), 16
`__delitem__()` (*encyclopedia.EAV method*), 18
`__getitem__()` (*encyclopedia.EAV method*), 18
`__init__()` (*encyclopedia.EAV method*), 18
`__init__()` (*encyclopedia.Relation method*), 19
`__invert__()` (*encyclopedia.Indexed method*), 16
`__invert__()` (*encyclopedia.Relation method*), 19
`__len__()` (*encyclopedia.Relation method*), 19
`__mul__()` (*encyclopedia.Unindexed method*), 15
`__neg__()` (*encyclopedia.Signed method*), 16
`__setitem__()` (*encyclopedia.EAV method*), 18
`__setitem__()` (*encyclopedia.Relation method*), 19
`__str__()` (*encyclopedia.EAV method*), 18
`__str__()` (*encyclopedia.Relation method*), 19
`__sub__()` (*encyclopedia.Unindexed method*), 15
`__weakref__` (*encyclopedia.EAV attribute*), 18

A

`above()` (*encyclopedia.Forest method*), 20
`aliased()` (*encyclopedia.Forest method*), 20
`Arboretum` (*class in encyclopedia*), 21
`attributes()` (*encyclopedia.EAV method*), 18

B

`below()` (*encyclopedia.Forest method*), 20
`branches()` (*encyclopedia.Forest method*), 20

C

`climb()` (*encyclopedia.Forest method*), 20
`compose()` (*encyclopedia.EAV method*), 18
`compose()` (*encyclopedia.Forest method*), 20
`compose()` (*encyclopedia.Relation method*), 19
`Coordinate` (*encyclopedia.KML attribute*), 22
`coordinated()` (*encyclopedia.KML static method*), 22
`copy()` (*encyclopedia.EAV method*), 18
`copy()` (*encyclopedia.Unindexed method*), 15
`copy()` (*encyclopedia.XML method*), 21
`copy_style()` (*encyclopedia.EAV method*), 18
`cutting()` (*encyclopedia.Forest method*), 20

D

`Dictionary` (*class in encyclopedia*), 17
`draw()` (*encyclopedia.KML method*), 22

E

`EAV` (*class in encyclopedia*), 17

F

`Forest` (*class in encyclopedia*), 20
`Forest.Node` (*class in encyclopedia*), 20
`freeze()` (*encyclopedia.Unindexed method*), 15

G

`graft()` (*encyclopedia.Forest method*), 20

H

`height()` (*encyclopedia.Forest method*), 20

I

`identified()` (*encyclopedia.Forest.Node method*), 20
`Indexed` (*class in encyclopedia*), 15
`instance()` (*encyclopedia.Record method*), 19

K

`KML` (*class in encyclopedia*), 22

L

`leaves()` (*encyclopedia.Forest method*), 21
`limb()` (*encyclopedia.Forest method*), 21

M

`melt()` (*encyclopedia.Unindexed method*), 15

P

`prune()` (*encyclopedia.Forest method*), 21

R

`Record` (*class in encyclopedia*), 18
`Relation` (*class in encyclopedia*), 19
`Relation.Error`, 19
`rename()` (*encyclopedia.EAV method*), 18
`root()` (*encyclopedia.Forest method*), 21

S

`Signed` (*class in encyclopedia*), 16
`sorted()` (*encyclopedia.Forest method*), 21
`sprout()` (*encyclopedia.Forest method*), 21
`Style` (*encyclopedia.KML attribute*), 22
`styled()` (*encyclopedia.KML static method*), 22
`stylize()` (*encyclopedia.KML method*), 22
`subtract()` (*encyclopedia.EAV method*), 18
`subtract()` (*encyclopedia.Forest method*), 21
`subtract()` (*encyclopedia.Relation method*), 19

U

`unfrozen()` (*encyclopedia.Forest method*), 21
`unfrozen()` (*encyclopedia.Relation method*), 19
`Unindexed` (*class in encyclopedia*), 15
`unique()` (*encyclopedia.XML method*), 21
`update()` (*encyclopedia.Forest method*), 21
`update()` (*encyclopedia.Relation method*), 19

V

`values()` (*encyclopedia.Forest method*), 21
`values()` (*encyclopedia.Relation method*), 19

W

`write()` (*encyclopedia.KML method*), 22
`write()` (*encyclopedia.XML method*), 21

X

`XML` (*class in encyclopedia*), 21