Relation: The Missing Container

Abstract

The humble mathematical relation, a fundamental (if implicit) component in computational algorithms, is conspicuously absent in most standard container sets, including Python's. We present the many uses of the relation and how you can use it to simplify your own coding.

Overview

A mathematical relation[1] when directly integrated into software can perform a variety of common tasks:

Inversion
: quickly find the values(range) associated with a key(domain)

Aliasing
: maintain a unique relationship between keys and values

Partitioning
: categorize values into unique groups

Tagging
: associate two sets in an arbitrary(bipartite) manner

When computational data is:

- well-structured
- vectorized
- large enough to be concerned about performance/storage

Existing computational [2] and relational [3] libraries can perform the above tasks, and do so very well at scale. However, when the data is:

- loosely structured or unstructured
- transient
- in no real danger of overloading memory or affecting performance

We tend to cobble together the containers we already have. This results in an ad hoc approach to each data structure, repeated code snippets, and repeated errors. This is unnecessary.

Using a relation container we can:

- reduce coding overhead
- increase clarity of purpose
- improve computational efficiency

Example 1 (Many-to-many)

Let's make some fruit:

```
fruit = Relation()
fruit['apple']='round'
fruit['pear']='yellow'
fruit['banana']='yellow'
fruit['banana']='crescent'
...
```

Note that we're overriding the assignment operator to allow for appending, not overwriting. So:

```
fruit['banana']
```

Produces a set of values:

```
{'yellow','crescent'}
```

Our relation also provides an inverse, so we can also get reverse look-up:

```
(~fruit)['yellow']
```

Also producing a set of values:

```
{'banana','pear'}
```

Cardinality

Relations provide the ability to

- assign many-to-many values
- invert the mapping

Relations become even more valuable when we have the ability to enforce the degree of relationship, i.e. cardinality. There are four cardinalities of relationship:

Relationship	Shortcut	Pseudonyms
many-to-one	M:1	function, mapping, assignment
one-to-many	1:M	partition
one-to-one	1:1	aliasing, isomorphism, invertible function
many-to-many	M:N	general relation

Assignment (M:1) is already supported[4] by Python's built-in dictionaries and lists; however, the remainder of the cardinalities are not[5].

Example 2 (One-to-One)

```
people = Relation(cardinality='1:1')
people['Fred'] = '111-00-6379'
people['Wilma'] = '222-00-7891'
```

When the relation is forced to be an isomorphism, the results are no longer sets:

```
people['Fred']
> '111-00-6389'
```

Assignments overwrite both the domain and the range:

```
people['Barney'] = '111-00-6379'
people['Fred']
> KeyError: 'Fred'
```

For cardinalities M:1 (*Function*) and 1:1 (*Isomorphism*), the standard dictionary can serve as an initializer:

```
people = Isomorphism(
    {'Barney':'111-00-6379',
    'Wilma':'222-00-7891'})
people['Barney']
> '111-00-6389'
```

And insertion order can be preserved (for all cardinalities) if desired:

```
people = Isomorphism(ordered=True)
people['Wilma'] = '222-00-7891'
people['Barney'] = '111-00-6379'

list(people.keys())
> ['Wilma','Barney']
```

More Examples

The relation is a basic concept, and as such useful in limitless contexts. Still, a few more example are worth mentioning.

Tags (Many-to-Many)

Over the last decade we've seen *tags* invade our previously hierarchical organized data. Tags are now ubiquitous, attached to our: photos, files, URL bookmarks, to-do items etc …

Tags are also exactly an M:N relationship:

```
files = Relation()

files['vacation-yellowstone.png'] = 'image'
files['vacation-yellowstone.png'] = 'family'
files['obnoxious-cat.jpeg'] = 'image'
files['vacation-planning.doc'] = 'family'
files['vacation-planning.doc'] = 2015

(~files)['family']
> {'vacation-yellowstone.png','vacation-planning.doc'}

files['vacation-planning.doc']
> {2015,'family'}
```

Taxonomies (One-to-Many)

Nesting partition relations (1:M) creates a backward-searchable taxonomy:

```
animals=Relation(cardinality='1:M')
animals['Mammal'] = 'Carnivore'
animals['Mammal'] = 'Monotreme'
animals['Monotreme'] = 'Duckbill Platypus'
animals['Monotreme'] = 'Spiny Anteater'
animals['Carnivore'] = 'Canine'
animals['Carnivore'] = 'Feline'
animals['Canine'] = 'Poodle'
animals['Canine'] = 'Labrador Retriever'
animals['Feline'] = 'Cat'


(~animals)['Poodle']
> 'Canine'

(~animals)[(~animals)['Poodle']]
> 'Carnivore'

(~animals)[(~animals)[(~animals)['Poodle']]]
> 'Mammal'
```

When to Use What for What

Modern high-level computing languages provide us with a wealth of containers. We feel, of course, that a relation container is a valuable addition but we also feel one should use the most economical and obvious container for the task. Asking questions about the type of data being stored and the relationship between an element and its attributes is crucial:

Structure	What to Use
unordered set of unique objects	set
ordered set of non-unique objects	list
ordered set of unique objects	OrderedDict
unidirectional mapping	dictionary
mapping with inversion	relation
mapping with restricted cardinalities	relation
multiple, fixed attributes per element	data frame/table
multiple, variant attributes per element	relation

Implementing the Relation Container

One of the best things about the relation data container is its ease of implementation within Python. See https://pypi.python.org/pypi/relate for an simple, yet complete, implementation as well as more information.

Conclusion

The relation provides an easy-to-use invertible mapping structure supporting all four relationship cardinalities. Using a relation can

simplify your code and eliminate the need for repeated, ad hoc patterns when managing your working data structures.

References

1. http://simple.wikipedia.org/wiki/Relation_(mathematics) ↩
2. numpy, pandas, etc… ↩
3. sqlite, postgress, etc… ↩
4. However, dictionaries do *not* have have the invertibility provide by a relation ↩
5. For 1:1 mapping, however we also recommend the excellent bidict package https://bidict.readthedocs.org/en/master/intro.html#intro ↩