rolne Documentation

Release 0.2.7

Maker Redux Corporation

October 15, 2014

Contents

rolne data type: Recursive Ordered Lists of Named Elements

Contents:

CHAPTER 1

An Overview of rolne

A **rolne** is a new data type that is *ultra-inclusive*. As such, it is a useful tool to interpret complex (and less-predictable) data sets such as XML documents, MARDS documents, and configuration files.

1.1 Comparing to Lists and Dictionaries

If you are already familiar with Python's dictionaries and lists, then the following might give useful insight into what I mean by *ultra-inclusive*:

A rolne is like a dictionary because you use a name to reference elements:

```
>>> zippy["size"] = 4
```

A rolne is like a list because the elements are _ordered_.

```
>>> zippy.append("fruit", "apple")
>>> zippy.append("fruit", "bannana")
>>> zippy.append("fruit", "orange")
>>> zippy.append("fruit", "bannana")
>>> zippy.list_values("fruit")
['apple', 'bannana', 'orange', 'bannana']
```

And can be referred to by an index:

```
>>> zippy.has("fruit", "bannana", 1)
True
```

But,

- In a dictionary, the names (keys) must be unique.

  In a rolne, multiple items can (and often do) have the same name.

- In a list, the place in the list is strictly an integer index.

  In a rolne, the place in the list is based on the name, value, *and* index.

1.2 Let's Start with an Example

Let's borrow one of the examples from XML docs at w3schools.com (http://www.w3schools.com/xml/xml_attributes.asp):

```
<messages>
  <note id="501">
    <to>Tove</to>
    <from>Jani</from>
    <heading>Reminder</heading>
    <body>Don't forget me this weekend!</body>
  </note>
  <note id="502">
    <to>Jani</to>
    <from>Tove</from>
    <heading>Re: Reminder</heading>
    <body>I will not</body>
  </note>
</messages>
```

If this were parsed into a rolne, it would end up looking like this

```
>>> my_xml
%rolne:
messages is None
    note is None
        id = 501
        to = Tove
        from = Jani
        heading = Reminder
        body = Don't forget me this weekend!
    note is None
        id = 502
        to = Jani
        from = Tove
        heading = Re: Reminder
        body = I will not
```

In this example, the *root* level of the **my_xml** has a single item in it:

```
>>> my_xml.list_keys()
[("messages", None, 0)]
```

Take note of the 'list' of keys: it contains one item. It has a **name** of `"messages"`, a **value** of `None`, and an **index** of zero (`0`). The index is zero because it is the *first* name/value pair to contain `"messages"`/`None`.

If you wanted to see the *children* of messages, one could reference it by key:

```
>>> my_xml["messages", None, 0]
%rolne:
note is None
    id = 501
    to = Tove
    from = Jani
    heading = Reminder
    body = Don't forget me this weekend!
note is None
    id = 502
    to = Jani
    from = Tove
    heading = Re: Reminder
    body = I will not
```

Notice that **all three parts** of the key where used. That is a means of explicitly identifying one of the children. However, as a conveniece, you don't have to use all three parts. So, the following works the same:

```
>>> my_xml["messages"]
%rolne:
note is None
    id = 501
    to = Tove
    from = Jani
    heading = Reminder
    body = Don't forget me this weekend!
note is None
    id = 502
    to = Jani
    from = Tove
    heading = Re: Reminder
    body = I will not
```

If not specified, the following are the base assumptions:

- name is <any>
- value is <any> (when locating, value is None when specifying)
- index is 0

So, `["messages"]` finds the first item with a name of *messages* regardless of the value.

Now, let's dive down further:

```
>>> my_xml["messages"]["note", None, 1]
%rolne:
id = 502
to = Jani
from = Tove
heading = Re: Reminder
body = I will not
```

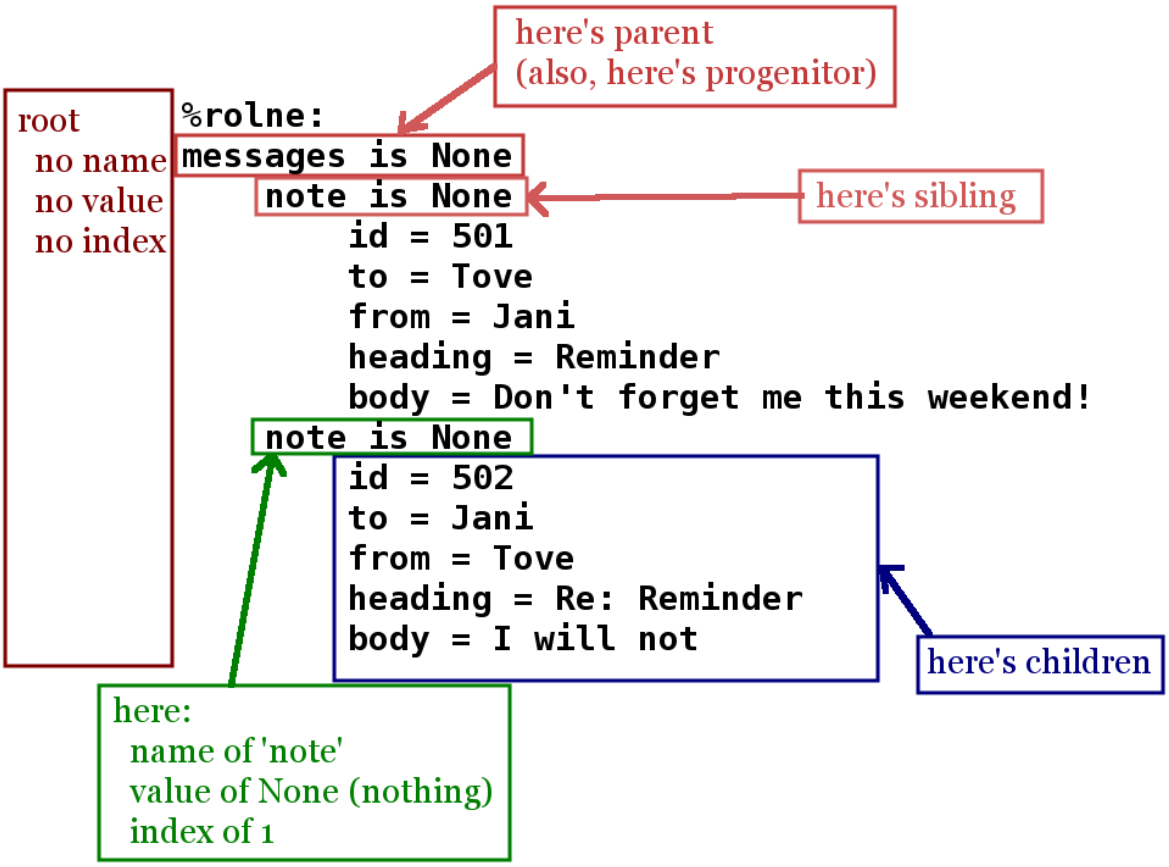
In this case, we are looking at the first `"messages"`/`None` and the second `"note"`/`None`. Notice the `1` index. That references the second item of any `"note"`/`None` items.

1.3 Relationships

Let me point a variable i'll call '**here**' at a specific location.

```
>> here = my_xml["messages"]["note", None, 1]
```

Now let's examine some of the relationships of the variable called "**here**":



Let's try some stuff out:

```
>>> here.name
note
>>> here.value
None
>>> here.index
1
>>> here.parents_name()
messages
>>> here.value("id")
502
>>> here.value("heading")
Re: Reminder
>>> here.list_values("to")
["Jani"]
>>> here.list_values("to")[0]
Jani
>>> here.find("to")
%rolne:
empty
>>> here.find("to").value
Jani
>>> here.find("to", "Jani", 0).value
Jani
>>> here["to"].value
```

```
Jani
```

BTW, what is the difference between `here.find("to")` and `here["to"]`. Allow me to demonstrate with a search for a key that does not exist:

```
>>> here["blah"]
KeyError: "('blah',) not found"
>>> here.find("blah")
None
```

Essentially, the *find* method avoids key errors by returning None rather than a subtending rolne.

Most of the expected behaviors one would expect from a pythonic class are supported. For example, iteration:

```
>>> for item in here:
...     print "name='{}', value='{}', index={}".format(item.name, item.value, item.index)
...
name='id', value='502', index=0
name='to', value='Jani', index=0
name='from', value='Tove', index=0
name='heading', value='Re: Reminder', index=0
name='body', value='I will not', index=0
```

And, of course, one can add/remove/update items:

```
>>> here["to"].value = "Steve"
>>> here["to"].value
Steve
>>> here.append("date", "2014-03-23")
>>> here.append("code", [0, 39, 2])
>>> del here["from"]
>>> print here
%rolne:
id = 502
to = Steve
heading = Re: Reminder
body = I will not
date = 2014-03-23
code = [0, 39, 2]
```

1.4 Added Bonus: Sequence References

In addition the basics, rolne also supports 'meta' sequences strings. Essentially, as each element is added rolne a new tracking string is also assigned to the name/value pair. One can simply ignore this. It is not critical to rolne's use. But it can be a useful short cut for remembering where "something" is.

One can see the sequences by using the `._explicit()` method:

```
>>> print here._explicit()
%rolne:
[19] id = 502
[20] to = Steve
[22] heading = Re: Reminder
[23] body = I will not
[32] date = 2014-03-23
[33] code = [0, 39, 2]
```

Some items to take note of:

- Don't try to "predict" the auto-numbering. You can only count on it's consistence within the context of a single rolne instance. There is no guarantee you will get the same numbering every time you run your program.
- Changing the **name** or **value** (or **index**) of an element does NOT change its sequence. The sequence is only set on insertion.

You can purposely set your own key. The rolne simply checks to make sure the seqence given is unique.

```
 >>> here.append("something", True, seq="hello")
 >>> print here._explicit()
%rolne:
[19] id = 502
[20] to = Steve
[22] heading = Re: Reminder
[23] body = I will not
[32] date = 2014-03-23
[33] code = [0, 39, 2]
[hello] something = True
```

1.5 Conclusion

You have just been given a quick summary. There is actaully far more to things than this. For example, one can:

- 'replace' child lines with other rolnes or child lines
- copy with prefix and suffix clauses for sequences
- list the lineage of any element

And lot's more. Have fun.

CHAPTER 2

rolne's Class Methods

class `rolne.rolne`(*in_list=None*, *in_tuple=None*, *ancestor=None*, *NS=None*)

append(*name*, *value=None*, *sublist=None*, *seq=None*)

Add one name/value entry to the main context of the rolne.

If you are wanting to "append" another rolne, see the 'extend' method instead.

Example of use:

```
>>> # setup an example rolne first
>>> my_var = rolne()
>>> my_var.append("item", "zing")
>>> my_var["item", "zing"].append("size", "4")
>>> my_var["item", "zing"].append("color", "red")
>>> print my_var
%rolne:
item = zing
    size = 4
    color = red

>>> my_var.append("item", "zing")
>>> my_var["item", "zing", -1].append("size", "2")
>>> my_var["item", "zing", -1].append("color", "blue")
>>> print my_var
%rolne:
item = zing
    size = 4
    color = red
item = zing
    size = 2
    color = blue
```

New in version 0.1.1.

Parameters

- **name** – The key name of the name/value pair.
- **value** – The key value of the name/value pair. If not passed, then the value is assumed to be None.
- **sublist** – An optional parameter that also appends a subtending list of entries. It is not recommended that this parameter be used.

append_index(*name*, *value=None*, *sublist=None*, *seq=None*)

Add one name/value entry to the main context of the rolne and return the index number for the new entry.

If you are wanting to "append" another rolne, see the 'extend' method instead.

Example of use:

```
>>> # setup an example rolne first
>>> my_var = rolne()
>>> index = my_var.append_index("item", "zing")
>>> print index
0
>>> my_var["item", "zing", index].append("size", "4")
>>> my_var["item", "zing", index].append("color", "red")
>>> print my_var
%rolne:
item = zing
    size = 4
    color = red

>>> index = my_var.append_index("item", "zing")
>>> print index
1
>>> my_var["item", "zing", index].append("size", "2")
>>> my_var["item", "zing", index].append("color", "blue")
>>> print my_var
%rolne:
item = zing
    size = 4
    color = red
item = zing
    size = 2
    color = blue
```

New in version 0.1.4.

Parameters

- **name** – The key name of the name/value pair.
- **value** – The key value of the name/value pair. If not passed, then the value is assumed to be None.
- **sublist** – An optional parameter that also appends a subtending list of entries. It is not recommended that this parameter be used.

Returns An integer representing the index of the newly inserted name/pair.

append_seq(*name*, *value=None*, *sublist=None*, *seq=None*)

Add one name/value entry to the current context of the rolne and return the new sequence string.

If you are wanting to "append" another rolne, see the 'extend' method instead.

Example of use:

```
>>> # setup an example rolne first
>>> my_var = rolne()
>>> my_var.append("item", "zing")
>>> my_var["item", "zing"].append("size", "4")
>>> my_var["item", "zing"].append("color", "red")
>>> print my_var
%rolne:
```

```
item = zing
    size = 4
    color = red

>>> my_var.append("item", "zing")
>>> my_var["item", "zing", -1].append("size", "2")
>>> my_var["item", "zing", -1].append("color", "blue")
>>> print my_var
%rolne:
item = zing
    size = 4
    color = red
item = zing
    size = 2
    color = blue
```

New in version 0.1.1.

Parameters

- **name** – The key name of the name/value pair.
- **value** – The key value of the name/value pair. If not passed, then the value is assumed to be None.
- **sublist** – An optional parameter that also appends a subtending list of entries. It is not recommended that this parameter be used.

find(**argv*)

Locate a single rolne entry.

This function is very similar to simply doing a dictionary-style lookup. For example:

new_rolne = my_var.find("test", "zoom", 4)

is effectively the same as:

new_rolne = my_var["test", "zoom", 4]

The biggest difference is that if entry at ["test", "zoom", 4] does not exist, the dictionary-style lookup generates a key error. Whereas this method simply returns None.

Example of use:

```
>>> # setup an example rolne first
>>> my_var = rolne()
>>> my_var.append("item", "zing")
>>> my_var["item", "zing"].append("size", "4")
>>> my_var["item", "zing"].append("color", "red")
>>> my_var["item", "zing"]["color", "red"].append("intensity", "44%")
>>> my_var["item", "zing"].append("reverse", None)
>>> my_var.append("item", "broom")
>>> my_var["item", "broom", -1].append("size", "1")
>>> my_var["item", "broom", -1].append("title", 'The "big" thing')
>>> my_var.append("item", "broom")
>>> my_var["item", "broom", -1].append("size", "2")
>>> my_var["item", "broom", -1].append("title", 'Other thing')
>>> my_var.append("code_seq")
>>> my_var["code_seq", None].append("*", "r9")
>>> my_var["code_seq", None].append("*", "r3")
>>> my_var["code_seq", None].append("*", "r2")
>>> my_var["code_seq", None].append("*", "r3")
```

```
>>> my_var.append("system_title", "hello")
>>> #
>>> print my_var.find("item","broom",1)
%rolne:
size = 2
title = "Other thing"

>>> print my_var.find("item","broom",2)
None
>>> print my_var["code_seq", None].find("*","r3")
%rolne:
%empty
```

New in version 0.1.2.

Parameters

- **name** – The key name of the name/value pair.
- **value** – The key value of the name/value pair. If not passed, then the value is assumed to be empty (None).
- **index** – The index of the name/value pair. if not passed, then the index is assumed to be 0.

Returns Returns either a rolne that points to the located entry or None if that entry is not found.

name

Name property.

This property represents the name of the rolne in its current context. For a new or original rolne, the name is always None. That is because the *root* of a rolne cannot conceptually have its own name.

It is possible to both read and write to the name property. Any change to name is also seen in other contexts. It is strongly recommended that name be given an immutable value.

It is not possible to delete the name property.

Example of use:

```
>>> # setup an example rolne first
>>> my_var = rolne()
>>> my_var.append("item", "zing")
>>> my_var["item", "zing"].append("size", "4")
>>> my_var["item", "zing"].append("color", "red")
>>> my_var["item", "zing"]["color", "red"].append("intensity", "44%")
>>> my_var["item", "zing"].append("reverse", None)
>>> my_var.append("item", "broom")
>>> my_var["item", "broom", -1].append("size", "1")
>>> my_var["item", "broom", -1].append("title", 'The "big" thing')
>>> my_var.append("item", "broom")
>>> my_var["item", "broom", -1].append("size", "2")
>>> my_var["item", "broom", -1].append("title", 'Other thing')
>>> #
>>> print my_var.name
None
>>> temp = my_var["item", "broom", 2]
>>> print temp.name
item
>>> temp.name = "hello"
>>> print temp.name
hello
```

```
>>> print my_var["hello", "broom", 2].name
hello
```

New in version 0.2.1.

seq

Sequence property.

This property represents the sequence assigned to the rolne in its current context. For a new or original rolne, the seq is always None. That is because the *root* of a rolne cannot conceptually have a sequence.

You can both read and write seq. However, it is not possible to delete. If written, the given value is always converted to a string. TODO: test string conversion

New in version 0.2.1.

upsert (*name*, *value=None*, *seq=None*)

Add one name/value entry to the main context of the rolne, but only if an entry with that name does not already exist.

If the an entry with name exists, then the first entry found has it's value changed.

NOTE: the upsert only updates the FIRST entry with the name found.

The method returns True if an insertion occurs, otherwise False.

Example of use:

```
>>> # setup an example rolne first
>>> my_var = rolne()
>>> my_var.upsert("item", "zing")
True
>>> my_var["item", "zing"].append("color", "blue")
>>> print my_var
%rolne:
item = zing
    color = blue

>>> my_var.upsert("item", "zing")
False
>>> print my_var
%rolne:
item = zing
    color = blue

>>> my_var.upsert("item", "broom")
False
>>> print my_var
%rolne:
item = broom
    color = blue
```

New in version 0.1.1.

Parameters

- **name** – The key name of the name/value pair.
- **value** – The key value of the name/value pair. If not passed, then the value is assumed to be None.

Returns Returns True if the name/value was newly inserted. Otherwise, it returns False indicated that an update was done instead.

value

value property.

This property represents the value (of name/value) assigned to the rolne in its current context. For a new or original rolne, the seq is always None. That is because the *root* of a rolne cannot conceptually have a value.

You can both read and write value. If it is deleted, it is simply set to None.

New in version 0.2.1.

r