voevent-parse

Release 1.0.3

Jun 24, 2018

Contents

Version 1.0.3

Welcome to voevent-parse's documentation. If you're new here, I recommend you start with the *introduction*. Then, take a look at the *usage examples*, or dive into the *tutorial*.

CHAPTER 1

Contents

1.1 Introduction

1.1.1 What is voevent-parse?

A lightweight library for parsing, manipulating, and generating VOEvent XML packets, built atop lxml.objectify and compatible with Python 2 and 3.

voevent-parse provides convenience routines to take care of many common tasks, so that accessing those vital data elements is as simple as:

```
import voeventparse
with open(xml_filename, 'rb') as f:
    v = voeventparse.load(f)
print "AuthorIVORN:", v.Who.AuthorIVORN  #Prints e.g. ivo://nasa.gsfc.tan/gcn
v.Who.AuthorIVORN = 'ivo://i.heart.python/lxml' #Alters the XML value.
```

1.1.2 Rationale

voevent-parse aims to make dealing with VOEvent packets easy, while remaining small, flexible, and stable enough to be suitable for use as a dependency in a range of larger projects. To achieve this, we add a user-friendly layer on top of lxml.objectify which attempts to hide the messy details of working with the sometimes lengthy VOEvent schema, and also take care of some rather obscure lxml namespace handling. However, since the objects created are just regular lxml classes, the user is free to utilise the full power of the lxml library when required.

1.1.3 Installation

voevent-parse is pip installable, try running:

```
pip install voevent-parse
```

Note that voevent-parse depends upon lxml, and pip will attempt to install lxml first if not already present. lxml may be installed as a system package if the version distributed with your package manager is sufficiently up-to-date (version >= 2.3). If you're working with pip / virtualenv and not making use of system packages, then note that lxml has some prerequisites for compilation that can cause a standard `pip install` to fail with somewhat cryptic errors. On a typical Debian / Ubuntu machine you can satisfy those requirements using:

```
sudo apt-get install libxml2-dev libxslt-dev
```

1.1.4 Documentation

Reference documentation can be found at http://voevent-parse.readthedocs.org, or generated directly from the repository using Sphinx.

1.1.5 Source, Issues, Development etc.

I intend to mark any updates by bumping the version number accordingly. That said, if you find yourself using voevent-parse in any serious context, do drop me an email so I can keep you informed of any updates or critical bugs.

Bug reports (or even better, pull requests) are welcomed. The source code and issue tracker may be found at https://github.com/timstaley/voevent-parse.

voevent-parse also has a suite of unit-tests which may be run in the usual manner, typically using nose from the repository root directory.

1.1.6 lxml.objectify 'gotchas'

Note: See also the tutorial, which includes a basic introduction to lxml.objectify.

The objectify library has a few syntactic quirks which can trip up new users. Firstly, you should be aware that the line `root.foo` actually returns an object that acts like a *list* of all the children of the `root` element with the name *foo*. What's confusing is that objectify has syntactic sugar applied so that `root.foo` is a shortcut alias for the more explicit `root.foo[0]`. This can be very confusing to the uninitiated, since it overrides some attributes of the the actual element values. To get around this, you should be aware of the accessor to the text representation of the value; `.text`, e.g.:

```
import lxml.objectify
root = lxml.objectify.Element('root')
root.foo = 'sometext' # Adds a child called 'foo' with value 'sometext'
print root.foo # 'sometext'
print len(root.foo) #  1. Wait, what?
# The string value clearly does not have length==1,
# the list of children called 'foo' does.
print root.foo.text # 'sometext'
print len(root.foo.text) # 8. Sanity prevails!
```

Another 'gotcha' is that *creating* multiple child elements of the same name is a bit unintuitive. Essentially, `objectify` works implicitly if each element has only one child:

```
from lxml import objectify, etree
simple_root = objectify.Element('simple_root')
simple_root.layer1 = None
```

(continues on next page)

(continued from previous page)

```
simple_root.layer1.layer2 = 5
print etree.tostring(simple_root, pretty_print=True)
```

But if there are multiple children then each child must be explicitly declared as an lxml `Element` in order to co-exist with its siblings:

```
from lxml import objectify, etree
import math
siblings_root = objectify.Element('siblings')
siblings_root.bars = None
siblings_root.bars.append(objectify.Element('bar'))
siblings_root.bars.append(objectify.Element('bar'))
siblings_root.bars.bar[0] = math.pi
siblings_root.bars.bar[1] = 42
print etree.tostring(siblings_root, pretty_print=True)
```

. . . which is another reason to use voevent-parse as a user-friendly interface for common operations.

For some more examples, you might also try: http://www.saltycrane.com/blog/2011/07/example-parsing-xml-lxml-objectify/.

1.1.7 See also

Brokers

In order to receive VOEvent packets, you will require a utility capable of connecting to the VOEvent backbone. Two such tools are Comet and Dakota.

Associated utility routines

Depending on what you want to use your VOEvents for, you may be interested fourpiskytools, which provides a minimum working example of a broker / event-handler setup, and basic routines for submitting VOEvents to a broker for publication.

Experienced users may also want to take a look at fourpisky-core, which is much less easy-to-read but provides extensive examples of handling VOEvent data for real-time alerting purposes.

Further information

The 4PiSky project page at https://4pisky.org/voevents/ provides links to more information on using VOEvents for scientific work, and other VOEvent related tools.

1.1.8 Acknowledgement

If you make use of voevent-parse in work leading to a publication, we ask that you cite the ASCL entry.

1.2 Usage examples

Note: These scripts give some quick syntax examples. See also the *Tutorial* notebooks, for a step-by-step introduction to using voevent-parse and working with XML data.

1.2.1 Basic data access and manipulation

You can also `download` this example or view it on Github.

```
#!/usr/bin/python
"""A quick usage example.

Once voeventparse is installed, this should tell you most of what you need to know
in order to start doing things with VOEvent packets.

The attributes are built from the structure of the XML file,
so the best way to understand where the variable names come from is to simply
open the XML packet in your favourite web browser and dig around.

See also:
* lxml documentation at http://lxml.de/objectify.html
* VOEvent standard at http://www.ivoa.net/documents/VOEvent/
* VOEvent schema file at http://www.ivoa.net/xml/VOEvent/VOEvent-v2.0.xsd
"""
from __future__ import print_function

import pprint

import copy
import voeventparse
from voeventparse.fixtures.datapaths import swift_bat_grb_pos_v2

pp = pprint.PrettyPrinter()

with open(swift_bat_grb_pos_v2, 'rb') as f:
    v = voeventparse.load(f)

# Basic attribute access
print("Ivorn:", v.attrib['ivorn'])
print("Role:", v.attrib['role'])
print("AuthorIVORN:", v.Who.AuthorIVORN)
print("Short name:", v.Who.Author.shortName)
print("Contact:", v.Who.Author.contactEmail)

# Copying by value, and validation:
print("Original valid as v2.0? ", voeventparse.valid_as_v2_0(v))
v_copy = copy.copy(v)
print("Copy valid? ", voeventparse.valid_as_v2_0(v_copy))

# Changing values:
v_copy.Who.Author.shortName = 'BillyBob'
v_copy.attrib['role'] = voeventparse.definitions.roles.test
print("Changes valid? ", voeventparse.valid_as_v2_0(v_copy))

v_copy.attrib['role'] = 'flying circus'
print("How about now? ", voeventparse.valid_as_v2_0(v_copy))
```

(continues on next page)

(continued from previous page)

```
print("But the original is ok, because we copied? ",
      voeventparse.valid_as_v2_0(v))

v.Who.BadPath = "This new attribute certainly won't conform with the schema."
assert voeventparse.valid_as_v2_0(v) == False
del v.Who.BadPath
assert voeventparse.valid_as_v2_0(v) == True
########################################################
# And now, SCIENCE
########################################################
c = voeventparse.get_event_position(v)
print("Coords:", c)
print()
toplevel_params = voeventparse.get_toplevel_params(v)

# print("Toplevel Params:")
# pp.pprint(toplevel_params.items())
print("Trigger ID:", toplevel_params['TrigID']['value'])
grouped_params = voeventparse.get_grouped_params(v)
# pp.pprint(grouped_params.allitems())
print(
"GRB Identified:", grouped_params['Solution_Status']['GRB_Identified']['value'])
```

1.2.2 Author a new VOEvent packet

You can also `download` this example or view it on Github.

```
#!/usr/bin/python
from __future__ import print_function

import datetime
import os

import pytz
import voeventparse as vp

# Set the basic packet ID and Author details

v = vp.Voevent(stream='astronomy.physics.science.org/super_exciting_events',
               stream_id=123, role=vp.definitions.roles.test)

vp.set_who(v, date=datetime.datetime.utcnow(),
           author_ivorn="voevent.4pisky.org")

vp.set_author(v, title="4PiSky Testing Node",
              shortName="Tim"
              )

# Now create some Parameters for entry in the 'What' section.

# Strictly speaking, parameter values should be strings,
# with a manually specified dataType; one of
# `string` (default), `int` , or `float`.
# e.g.
int_flux = vp.Param(name='int_flux',
```

(continues on next page)

(continued from previous page)

```
                    value="2.0e-3",
                    unit='Janskys',
                    ucd='em.radio.100-200MHz',
                    dataType='float',
                    ac=False)
int_flux.Description = 'Integrated Flux'

# But with ac=True (autoconvert) we switch on some magic to take care
# of this for us automatically.
# See ``Param`` docstring for details.
p_flux = vp.Param(name='peak_flux',
                  value=1.5e-3,
                  unit='Janskys',
                  ucd='em.radio.100-200MHz',
                  ac=True
                  )
p_flux.Description = 'Peak Flux'

v.What.append(vp.Group(params=[p_flux, int_flux], name='source_flux'))

# Note ac=True (autoconvert) is the default setting if dataType=None (the default)
amb_temp = vp.Param(name="amb_temp",
                    value=15.5,
                    unit='degrees',
                    ucd='phys.temperature')

amb_temp.Description = "Ambient temperature at telescope"
v.What.append(amb_temp)

# Now we set the sky location of our event:
vp.add_where_when(v,
                  coords=vp.Position2D(ra=123.5, dec=45, err=0.1,
                                       units='deg',
                                       system=vp.definitions.sky_coord_system.utc_fk5_
↪geo),
                  obs_time=datetime.datetime(2013, 1, 31, 12, 5, 30,
                                             tzinfo=pytz.utc),
                  observatory_location=vp.definitions.observatory_location.geosurface)

# Prettyprint some sections for desk-checking:
print("\n***Here is your WhereWhen:***\n")
print(vp.prettystr(v.WhereWhen))

print("\n***And your What:***\n")
print(vp.prettystr(v.What))

# You would normally describe or reference your telescope / instrument here:
vp.add_how(v, descriptions='Discovered via 4PiSky',
           references=vp.Reference('http://4pisky.org'))

# The 'Why' section is optional, allows for speculation on probable
# astrophysical cause
vp.add_why(v, importance=0.5,
           inferences=vp.Inference(probability=0.1,
                                   relation='identified',
                                   name='GRB121212A',
                                   concept='process.variation.burst;em.radio')
```

(continues on next page)

(continued from previous page)

```
            )

# We can also cite earlier VOEvents:
vp.add_citations(v,
                 vp.EventIvorn(
                     ivorn='ivo://astronomy.physics.science.org/super_exciting_events
↪#101',
                     cite_type=vp.definitions.cite_types.followup))

# Check everything is schema compliant:
vp.assert_valid_as_v2_0(v)

output_filename = 'new_voevent_example.xml'
with open(output_filename, 'wb') as f:
    vp.dump(v, f)

print("Wrote your voevent to ", os.path.abspath(output_filename))
```

1.3 Tutorial

1.3.1 Parsing VOEvent XML packets

Getting started

```
In [1]: from __future__ import print_function
        import voeventparse as vp
```

IPython Tip #1: In IPython (terminal *or* notebook) you can quickly check the docstring for something by putting a question mark in front, e.g.

```
In [2]: # Uncomment the following and hit enter:
        # ?vp.load
```

Alternatively, you can always read the docs, which include autogenerated API specs.

Ok, let's load up a voevent (click here to see the raw XML in your browser):

```
In [3]: with open('voevent.xml','rb') as f:
            v = vp.load(f)
```

IPython Tip #2: We also get tab-completion. Simply start typing the name of a function (or even just the '.' operator) and hit tab to see valid possible options - this is handy for exploring VOEvent packets:

```
In [4]: # Uncomment the following and hit tab:
        # v.W
```

Accessing data

Text-values

XML Tip #1: An XML packet is a tree-structure made composed of elements. We can dig into the tree structure of the VOEvent, and inspect values:

```
In [5]: # v.Who.Date.text
        # vp.pull_isotime(v)
```

```
In [6]: print("Inferred reason is", v.Why.Inference.Name.text)
        print( "(A string of length {})".format(len(v.Why.Inference.Name.text)))
        type(v.Why.Inference.Name.text)
```

```
Inferred reason is GRB121212A
(A string of length 10)
```

```
Out[6]: str
```

Attributes

XML Tip #2: Note that there are two ways to store data in an XML packet: * A single string can be stored as an element's text-value - like the two we just saw. * Alternatively, we can attach a number of key-value strings to an element, storing them as attributes. We can access these via `attrib`, which behaves like a Python dictionary, e.g.:

```
In [7]: print(v.attrib['ivorn'])
        print(v.attrib['role'])
```

```
ivo://example/exciting_events#123
test
```

```
In [8]: v.Why.Inference.attrib
```

```
Out[8]: {'relation': 'identified', 'probability': '0.1'}
```

```
In [9]: print(vp.prettystr(v.Who))
```

```
<Who>
  <AuthorIVORN>ivo://hotwired.org</AuthorIVORN>
  <Date>1970-01-01T00:00:00</Date>
  <Author>
    <title>Hotwired VOEvent Tutorial</title>
  </Author>
</Who>
```

'Sibling' elements and list-style access

So far, each of the elements we've accessed has been the only one of that name - i.e. our VOEvent has only one `Who` child-element, likewise there's only one `Inference` under the `Why` entry in this particular packet. But that's not always the case; for example the `What` section contains a `Group` with two child-elements called `Param`:

```
In [10]: print(vp.prettystr(v.What.Group))
```

```
<Group name="source_flux">
  <Param dataType="float" name="peak_flux" ucd="em.radio.100-200MHz" unit="Janskys" value="0.0015">
    <Description>Peak Flux</Description>
  </Param>
  <Param dataType="float" name="int_flux" ucd="em.radio.100-200MHz" unit="Janskys" value="2.0e-3">
    <Description>Integrated Flux</Description>
  </Param>
</Group>
```

So how do we access all of these? This is where we start getting into the details of lxml.objectify syntax (which voevent-parse uses under the hood). **lxml.objectify uses a neat, but occasionally confusing, trick: when we access a child-element by name, what's returned behaves like a list**:

```
In [11]: v.What[0] # v.What behaves like a list!
```

```
Out[11]: <Element What at 0x7fdbbb8e23c8>
```

However, to save having to type something like `v.foo[0].bar[0].baz[0]`, the first element of the list can also be accessed without the `[0]` operator (aka 'syntactic sugar'):

```
In [12]: v.What is v.What[0]
```

```
Out[12]: True
```

Knowing that it's 'just a list', we have a couple of options, we can iterate:

```
In [13]: for par in v.What.Group.Param:
             print(par.Description)
```

```
Peak Flux
Integrated Flux
```

Or we can check the length, access elements by index, etc:

```
In [14]: len(v.What.Group.Param)
```

```
Out[14]: 2
```

```
In [15]: v.What.Group.Param[1].Description
```

```
Out[15]: 'Integrated Flux'
```

Note that another example of this 'syntactic sugar' is that we can display the text-value of an element without adding the `.text` suffix.

However, see below for why it's a good idea to always use `.text` when you really do want the text-value of an element:

```
In [16]: print(v.Why.Inference.Name)  # More syntax sugar - if it has a string-value but no children,
         print(v.Why.Inference.Name.text) # The safe option
         print(v.Why.Inference.Name.text[:3]) # Indexing on the string as you'd expect
         print(v.Why.Inference.Name[:3]) # This is indexing on the *list of elements*, not the string
```

```
GRB121212A
GRB121212A
GRB
['GRB121212A']
```

If that all sounds awfully messy, help is at hand: you're most likely to encounter sibling elements under the `What` entry of a VOEvent, and voevent-parse has a pair of functions to convert that to nested dictionary-like structures for you:

```
In [17]: # Consult the docstring
         # ?vp.get_toplevel_params
         # ?vp.get_grouped_params
```

```
In [18]: grouped_params = vp.get_grouped_params(v)
         # what_dict
         list(grouped_params['source_flux'].items())
```

```
Out[18]: [('peak_flux',
           {'value': '0.0015', 'name': 'peak_flux', 'ucd': 'em.radio.100-200MHz', 'dataType': 'float
          ('int_flux',
           {'value': '2.0e-3', 'name': 'int_flux', 'ucd': 'em.radio.100-200MHz', 'dataType': 'float',
```

```
In [19]: grouped_params['source_flux']['peak_flux']['value']
```

```
Out[19]: '0.0015'
```

Advanced usage

Since voevent-parse uses lxml.objectify, the full power of the LXML library is available when handling VOEvents loaded with voevent-parse.

Iterating over child-elements

We already saw how you can access a group of child-elements by name, in list-like fashion. But you can also iterate over all the children of an element, even if you don't know the names ('tags', in XML-speak) ahead of time:

```
In [20]: for child in v.Who.iterchildren():
             print(child.tag, child.text, child.attrib)
```

```
AuthorIVORN ivo://hotwired.org {}
Date 1970-01-01T00:00:00 {}
Author None {}
```

```
In [21]: for child in v.WhereWhen.ObsDataLocation.ObservationLocation.iterchildren():
             print(child.tag, child.text, child.attrib)
```

```
AstroCoordSystem None {'id': 'UTC-FK5-GEO'}
AstroCoords None {'coord_system_id': 'UTC-FK5-GEO'}
```

Querying a VOEvent

Another powerful technique is to find elements using Xpath or ElementPath queries, but this is beyond the scope of this tutorial: we leave you with just a single example:

```
In [22]: v.find(".//Param[@name='int_flux']").attrib['value']
```

```
Out[22]: '2.0e-3'
```

Final words

Congratulations! You should now be able to extract data from just about any VOEvent packet. Note that voevent-parse comes with a few convenience routines to help with common, tedious operations, but you can always compose your own.

If you put together something that you think others could use (or find a bug!), pull requests are welcome.

Next stop: *authoring your own VOEvent*.

1.3.2 Authoring VOEvent packets

```
In [1]: from __future__ import print_function
        import voeventparse as vp
        import datetime
```

(To get started reading VOEvents, see the *previous notebook*).

Packet creation

We'll start by creating the skeleton of our VOEvent packet. We set the role to test so that nobody is tempted to start acting on the contents of this demo event. We also set the timestamp in the `Who` block to the time the event was generated (not when the observation was made), as per the specification:

```
In [2]: v = vp.Voevent(stream='hotwired.org/gaia_demo', stream_id=1,
                               role=vp.definitions.roles.test)
```

```
In [3]: #Set Who.Date timestamp to date of packet-generation:
        vp.set_who(v, date=datetime.datetime.utcnow(),
                author_ivorn="foo.hotwired.hotwireduniverse.org/bar")
```

```
        vp.set_author(v, title="Hotwired VOEvent Hands-on",
                                contactName="Joe Bloggs")
        v.Description = "This is not an official Gaia data product."
```

At any time, you can use `vp.dumps` (dump-string) to take a look at the VOEvent you've composed so far:

```
In [4]: # print(vp.dumps(v, pretty_print=True))
```

However, that's pretty dense! Use `vp.prettystr` to view a single element, which is a bit easier on the eyes:

```
In [5]: print(vp.prettystr(v.Who))
```

```
<Who>
  <Description>VOEvent created with voevent-parse, version 1.0.3. See https://github.com/timstaley/vo
  <AuthorIVORN>ivo://foo.hotwired.hotwireduniverse.org/bar</AuthorIVORN>
  <Date>2018-06-24T18:52:12</Date>
  <Author>
    <title>Hotwired VOEvent Hands-on</title>
    <contactName>Joe Bloggs</contactName>
  </Author>
</Who>
```

Adding `What` content

We'll add details from this GAIA event:

Name	UTC timestamp	RA	Dec	AlertMag	HistMag	HistStd-Dev	Class	Comment	Published
Gaia14adi	2014-11-07 01:05:09	168.47841	-23.01221	18.77	19.62	0.07	unknown	Fading source on top of 2MASS Galaxy (offset from bulge)	2 Dec 2014, 13:55

Now let's add details of the observation itself. We'll record both the magnitude that Gaia is reporting for this particular event, and the historic values they also provide:

```
In [6]: v.What.append(vp.Param(name="mag", value=18.77, ucd="phot.mag"))
        h_m = vp.Param(name="hist_mag", value=19.62, ucd="phot.mag")
        h_s = vp.Param(name="hist_scatter", value=0.07, ucd="phot.mag")
        v.What.append(vp.Group(params=[h_m, h_s], name="historic"))
```

Adding `WhereWhen` details

Now we need to specify where and when the observation was made. Rather than trying to specify a position for Gaia, we'll just call it out by name. Note that Gaia don't provide errors on the position they cite, so we're rather optimistically using 0:

```
In [7]: import pytz
        vp.add_where_when(v,
                          coords=vp.Position2D(ra=168.47841, dec=-23.01221, err=0, units='deg',
                                               system=vp.definitions.sky_coord_system.utc_fk5_geo),
                          obs_time=datetime.datetime(2014, 11, 7, 1, 5, 9, tzinfo=pytz.UTC),
                          observatory_location="Gaia")
```

```
In [8]: ## See how much element creation that routine just saved us(!):
        print(vp.prettystr(v.WhereWhen))
```

```
<WhereWhen>
  <ObsDataLocation>
    <ObservatoryLocation id="Gaia"/>
    <ObservationLocation>
      <AstroCoordSystem id="UTC-FK5-GEO"/>
      <AstroCoords coord_system_id="UTC-FK5-GEO">
        <Time unit="s">
          <TimeInstant>
            <ISOTime>2014-11-07T01:05:09</ISOTime>
          </TimeInstant>
        </Time>
        <Position2D unit="deg">
          <Name1>RA</Name1>
          <Name2>Dec</Name2>
          <Value2>
            <C1>168.47841</C1>
            <C2>-23.01221</C2>
          </Value2>
          <Error2Radius>0</Error2Radius>
        </Position2D>
      </AstroCoords>
    </ObservationLocation>
  </ObsDataLocation>
</WhereWhen>
```

Adding the `How`

We should also describe how this transient was detected, and refer to the name that Gaia have assigned it. Note that we can provide multiple descriptions (and/or references) here:

```
In [9]: vp.add_how(v, descriptions=['Scraped from the Gaia website',
                                    'This is Gaia14adi'],
                   references=vp.Reference("http://gsaweb.ast.cam.ac.uk/alerts/"))
```

And finally, `Why`

Finally, we can provide some information about why this even might be scientifically interesting. Gaia haven't provided a classification, but we can at least incorporate the textual description:

```
In [10]: vp.add_why(v)
         v.Why.Description = "Fading source on top of 2MASS Galaxy (offset from bulge)"
```

Check and save

Finally - and importantly, as discussed in the VOEvent notes - let's make sure that this event is really valid according to our schema:

```
In [11]: vp.valid_as_v2_0(v)
```

```
Out[11]: True
```

Great! We can now save it to disk:

```
In [12]: with open('my_gaia.xml', 'wb') as f:
             vp.dump(v, f)
```

And we're all done. You can open the file in your favourite text editor to see what we've produced, but note that it probably won't be particularly elegantly formatted - an alternative option is to open it in your browser.

Advanced Usage

Free-style element authoring

Note that if you want to do something that's not part of the standard use-cases addressed by voevent-parse, you can always use the underlying lxml.objectify tools to manipulate elements yourself. For example - don't like the 'voevent-parse' tag that gets added to your VOEvent Who skeleton? You can delete it:

```
In [13]: ## Before deletion:
         ## (Enclosed in an if-clause in case this is re-run after the cell below)
         if hasattr(v.Who, 'Description'):
             print(v.Who.Description)
```

```
VOEvent created with voevent-parse, version 1.0.3. See https://github.com/timstaley/voevent-parse fo
```

```
In [14]: if hasattr(v.Who, 'Description'):
             del v.Who.Description
         #Now it's gone!
         print(vp.prettystr(v.Who))
```

```
<Who>
  <AuthorIVORN>ivo://foo.hotwired.hotwireduniverse.org/bar</AuthorIVORN>
  <Date>2018-06-24T18:52:12</Date>
  <Author>
    <title>Hotwired VOEvent Hands-on</title>
    <contactName>Joe Bloggs</contactName>
  </Author>
</Who>
```

Want to add some additional elements of your own? Here's how, but: make sure you stick to the VOEvent schema!

```
In [15]: import lxml.objectify as objectify
```

```
In [16]: # This won't last long:
         vp.valid_as_v2_0(v)
```

```
Out[16]: True
```

If you just want a single text-value element (with no siblings of the same name), you can take a syntactic shortcut and simply assign to it:

(Under the hood this assigns a new child ''StringElement'' to that tag - if there are pre-existing elements with that same tag, it is assigned to the first position in the list, overwriting anything already there.)

```
In [17]: v.What.shortcut='some text assigned in a quick-and-dirty fashion'
         print (vp.prettystr(v.What))
```

```
<What>
  <Param dataType="float" name="mag" ucd="phot.mag" value="18.77"/>
  <Group name="historic">
    <Param dataType="float" name="hist_mag" ucd="phot.mag" value="19.62"/>
    <Param dataType="float" name="hist_scatter" ucd="phot.mag" value="0.07"/>
  </Group>
  <shortcut>some text assigned in a quick-and-dirty fashion</shortcut>
</What>
```

In general, though, you probably want to use SubElement, as this allows you to create multiple sibling-child elements of the same name, etc.

```
In [18]: for i in range(5):
             objectify.SubElement(v.What, 'foo')
             v.What.foo[-1]="foo{}".format(i)
         print("I have {} foos for you:".format(len(v.What.foo)))
         print (vp.prettystr(v.What))
```

```
I have 5 foos for you:
<What>
  <Param dataType="float" name="mag" ucd="phot.mag" value="18.77"/>
  <Group name="historic">
    <Param dataType="float" name="hist_mag" ucd="phot.mag" value="19.62"/>
    <Param dataType="float" name="hist_scatter" ucd="phot.mag" value="0.07"/>
  </Group>
  <shortcut>some text assigned in a quick-and-dirty fashion</shortcut>
  <foo>foo0</foo>
  <foo>foo1</foo>
  <foo>foo2</foo>
  <foo>foo3</foo>
  <foo>foo4</foo>
</What>
```

```
In [19]: # Get rid of all the foo:
         del v.What.foo[:]
```

Alternatively, you can create elements independently, then append them to a parent (remember how lxml.objectify pretends elements are lists?) - this is occasionally useful if you want to write a function that returns an element, e.g. to create a new Param (but voevent-parse wraps that up for you already):

```
In [20]: temp = objectify.Element('NonSchemaCompliantParam', attrib={'somekey':'somevalue'})
         v.What.append(temp)
         print(vp.prettystr(v.What))
```

```
<What>
  <Param dataType="float" name="mag" ucd="phot.mag" value="18.77"/>
  <Group name="historic">
    <Param dataType="float" name="hist_mag" ucd="phot.mag" value="19.62"/>
    <Param dataType="float" name="hist_scatter" ucd="phot.mag" value="0.07"/>
  </Group>
  <shortcut>some text assigned in a quick-and-dirty fashion</shortcut>
  <NonSchemaCompliantParam somekey="somevalue"/>
</What>
```

Obviously, these are non-schema compliant elements. Don't make up your own format - use Params for storing general data:

```
In [21]: vp.valid_as_v2_0(v)
```

```
Out[21]: False
```

Note that you can get a traceback if you need to figure out why a VOEvent is non-schema-compliant - this will report the first invalid element it comes across:

```
In [22]: try:
             vp.assert_valid_as_v2_0(v)
         except Exception as e:
             print(e)
```

```
Element 'shortcut': This element is not expected.
```

1.4 voevent-parse API reference

Warning: Much of the content within assumes the reader has at least a summary understanding of the VOEvent specifications.

Note: The top-level `__init__.py` file imports key classes and subroutines into the top-level `voeventparse` namespace, for brevity.

1.4.1 `voeventparse.voevent` - Basic VOEvent packet manipulation

Routines for handling etrees representing VOEvent packets.

`voeventparse.voevent.`**`Voevent`**(*stream*, *stream_id*, *role*)

Create a new VOEvent element tree, with specified IVORN and role.

Parameters

- **`stream`** (*`str`*) – used to construct the IVORN like so:

```
ivorn = 'ivo://' + stream + '#' + stream_id
```

(N.B. `stream_id` is converted to string if required.) So, e.g. we might set:

```
stream='voevent.soton.ac.uk/super_exciting_events'
stream_id=77
```

- **`stream_id`** (*`str`*) – See above.
- **`role`** (*`str`*) – role as defined in VOEvent spec. (See also *`definitions.roles`*)

Returns Root-node of the VOEvent, as represented by an lxml.objectify element tree ('etree'). See also http://lxml.de/objectify.html#the-lxml-objectify-api

`voeventparse.voevent.`**`loads`**(*s*, *check_version=True*)

Load VOEvent from bytes.

This parses a VOEvent XML packet string, taking care of some subtleties. For Python 3 users, `s` should be a bytes object - see also http://lxml.de/FAQ.html, "Why can't lxml parse my XML from unicode strings?" (Python 2 users can stick with old-school `str` type if preferred)

By default, will raise an exception if the VOEvent is not of version 2.0. This can be disabled but voevent-parse routines are untested with other versions.

Parameters

- **`s`** (*`bytes`*) – Bytes containing raw XML.
- **`check_version`** (*`bool`*) – (Default=True) Checks that the VOEvent is of a supported schema version - currently only v2.0 is supported.

Returns Root-node of the etree.

Return type *`Voevent`*

Raises `ValueError` – If passed a VOEvent of wrong schema version (i.e. schema 1.1)

voeventparse.voevent.**load**(*file*, *check_version=True*)

Load VOEvent from file object.

A simple wrapper to read a file before passing the contents to *loads()*. Use with an open file object, e.g.:

```
with open('/path/to/voevent.xml', 'rb') as f:
    v = vp.load(f)
```

Parameters

- **file** (*io.IOBase*) – An open file object (binary mode preferred), see also http://lxml.de/FAQ.html : "Can lxml parse from file objects opened in unicode/text mode?"
- **check_version** (*bool*) – (Default=True) Checks that the VOEvent is of a supported schema version - currently only v2.0 is supported.

Returns Root-node of the etree.

Return type *Voevent*

voeventparse.voevent.**dumps**(*voevent*, *pretty_print=False*, *xml_declaration=True*, *encoding='UTF-8'*)

Converts voevent to string.

Note: Default encoding is UTF-8, in line with VOE2.0 schema. Declaring the encoding can cause diffs with the original loaded VOEvent, but I think it's probably the right thing to do (and lxml doesn't really give you a choice anyway).

Parameters

- **voevent** (*Voevent*) – Root node of the VOevent etree.
- **pretty_print** (*bool*) – indent the output for improved human-legibility when possible. See also: http://lxml.de/FAQ.html#why-doesn-t-the-pretty-print-option-reformat-my-xml-output
- **xml_declaration** (*bool*) – Prepends a doctype tag to the string output, i.e. something like `<?xml version='1.0' encoding='UTF-8'?>`

Returns Bytestring containing raw XML representation of VOEvent.

Return type bytes

voeventparse.voevent.**dump**(*voevent*, *file*, *pretty_print=True*, *xml_declaration=True*)

Writes the voevent to the file object.

e.g.:

```
with open('/tmp/myvoevent.xml','wb') as f:
    voeventparse.dump(v, f)
```

Parameters

- **voevent** (*Voevent*) – Root node of the VOevent etree.
- **file** (*io.IOBase*) – An open (binary mode) file object for writing.
- **pretty_print** (*bool*) –
- **pretty_print** – See *dumps()*

- **xml_declaration** (*bool*) – See *dumps()*

voeventparse.voevent.**valid_as_v2_0**(*voevent*)
Tests if a voevent conforms to the schema.

Parameters **voevent** (*Voevent*) – Root node of a VOEvent etree.

Returns Whether VOEvent is valid

Return type bool

voeventparse.voevent.**assert_valid_as_v2_0**(*voevent*)
Raises `lxml.etree.DocumentInvalid` if voevent is invalid.

Especially useful for debugging, since the stack trace contains a reason for the invalidation.

Parameters **voevent** (*Voevent*) – Root node of a VOEvent etree.

Raises `lxml.etree.DocumentInvalid` – if VOEvent does not conform to schema.

voeventparse.voevent.**set_who**(*voevent*, *date=None*, *author_ivorn=None*)
Sets the minimal 'Who' attributes: date of authoring, AuthorIVORN.

Parameters

- **voevent** (*Voevent*) – Root node of a VOEvent etree.
- **date** (*datetime.datetime*) – Date of authoring. NB Microseconds are ignored, as per the VOEvent spec.
- **author_ivorn** (*str*) – Short author identifier, e.g. `voevent.4pisky.org/ALARRM`. Note that the prefix `ivo://` will be prepended internally.

voeventparse.voevent.**set_author**(*voevent*, *title=None*, *shortName=None*, *logoURL=None*, *contactName=None*, *contactEmail=None*, *contactPhone=None*, *contributor=None*)
For setting fields in the detailed author description.

This can optionally be neglected if a well defined AuthorIVORN is supplied.

Note: Unusually for this library, the args here use CamelCase naming convention, since there's a direct mapping to the `Author.*` attributes to which they will be assigned.

Parameters **voevent** (*Voevent*) – Root node of a VOEvent etree. The rest of the arguments are strings corresponding to child elements.

voeventparse.voevent.**add_where_when**(*voevent*, *coords*, *obs_time*, *observatory_location*, *allow_tz_naive_datetime=False*)
Add details of an observation to the WhereWhen section.

We

Parameters

- **voevent** (*Voevent*) – Root node of a VOEvent etree.
- **coords** (*Position2D*) – Sky co-ordinates of event.
- **obs_time** (*datetime.datetime*) – Nominal DateTime of the observation. Must either be timezone-aware, or should be carefully verified as representing UTC and then set parameter `allow_tz_naive_datetime=True`.

- **observatory_location** (*str*) – Telescope locale, e.g. 'La Palma'. May be a generic location as listed under *voeventparse.definitions.observatory_location*.
- **allow_tz_naive_datetime** (*bool*) – (Default False). Accept timezone-naive datetime-timestamps. See comments for `obs_time`.

voeventparse.voevent.**add_how**(*voevent*, *descriptions=None*, *references=None*)
Add descriptions or references to the How section.

Parameters

- **voevent** (*Voevent*) – Root node of a VOEvent etree.
- **descriptions** (*str*) – Description string, or list of description strings.
- **references** (*voeventparse.misc.Reference*) – A reference element (or list thereof).

voeventparse.voevent.**add_why**(*voevent*, *importance=None*, *expires=None*, *inferences=None*)
Add Inferences, or set importance / expires attributes of the Why section.

Note: `importance` / `expires` are 'Why' attributes, therefore setting them will overwrite previous values. `inferences`, on the other hand, are appended to the list.

Parameters

- **voevent** (*Voevent*) – Root node of a VOEvent etree.
- **importance** (*float*) – Value from 0.0 to 1.0
- **expires** (*datetime.datetime*) – Expiration date given inferred reason (See voevent spec).
- **inferences** (*voeventparse.misc.Inference*) – Inference or list of inferences, denoting probable identifications or associations, etc.

voeventparse.voevent.**add_citations**(*voevent*, *event_ivorns*)
Add citations to other voevents.

The schema mandates that the 'Citations' section must either be entirely absent, or non-empty - hence we require this wrapper function for its creation prior to listing the first citation.

Parameters

- **voevent** (*Voevent*) – Root node of a VOEvent etree.
- **event_ivorns** (*voeventparse.misc.EventIvorn*) – List of EventIvorn elements to add to citation list.

1.4.2 voeventparse.misc - Subtree-elements and other helpers

Routines for creating sub-elements of the VOEvent tree, and a few other helper classes.

class voeventparse.misc.**Position2D**
A namedtuple for simple representation of a 2D position as described by the VOEvent spec.

Parameters

- **ra** (*float*) – Right ascension.

- **dec** (*float*) – Declination
- **err** (*float*) – Error radius.
- **units** (*str*) – Coordinate units, cf *definitions.units* e.g. degrees, radians.
- **system** (*str*) – Co-ordinate system, e.g. UTC-FK5-GEO cf *definitions.sky_coord_system*

voeventparse.misc.**Param**(*name*, *value=None*, *unit=None*, *ucd=None*, *dataType=None*, *utype=None*, *ac=True*)

'Parameter', used as a general purpose key-value entry in the 'What' section.

May be assembled into a *Group*.

NB `name` is not mandated by schema, but *is* mandated in full spec.

Parameters

- **value** (*str*) – String representing parameter value. Or, if `ac` is true, then 'autoconversion' is attempted, in which case `value` can also be an instance of one of the following:
  - `bool`
  - `int`
  - `float`
  - `datetime.datetime`

  This allows you to create Params without littering your code with string casts, or worrying if the passed value is a float or a string, etc. NB the value is always *stored* as a string representation, as per VO spec.
- **unit** (*str*) – Units of value. See *definitions.units*
- **ucd** (*str*) – unified content descriptor. For a list of valid UCDs, see: http://vocabularies.referata.com/wiki/Category:IVOA_UCD.
- **dataType** (*str*) – Denotes type of `value`; restricted to 3 options: `string` (default), `int` , or `float`. (NB *not* to be confused with standard XML Datatypes, which have many more possible values.)
- **utype** (*str*) – See http://wiki.ivoa.net/twiki/bin/view/IVOA/Utypes
- **ac** (*bool*) – Attempt automatic conversion of passed `value` to string, and set `dataType` accordingly (only attempted if `dataType` is the default, i.e. `None`). (NB only supports types listed in _datatypes_autoconversion dict)

voeventparse.misc.**Group**(*params*, *name=None*, *type=None*)

Groups together Params for adding under the 'What' section.

Parameters

- **params** (list of *Param()*) – Parameter elements to go in this group.
- **name** (*str*) – Group name. NB `None` is valid, since the group may be best identified by its type.
- **type** (*str*) – Type of group, e.g. 'complex' (for real and imaginary).

voeventparse.misc.**Reference**(*uri*, *meaning=None*)

Represents external information, typically original obs data and metadata.

Parameters

- **uri** (*str*) – Uniform resource identifier for external data, e.g. FITS file.

- **meaning** (*str*) – The nature of the document referenced, e.g. what instrument and filter was used to create the data?

voeventparse.misc.**Inference**(*probability=None*, *relation=None*, *name=None*, *concept=None*)
Represents a probable cause / relation between this event and some prior.

Parameters

- **probability** (*float*) – Value 0.0 to 1.0.
- **relation** (*str*) – e.g. 'associated' or 'identified' (see Voevent spec)
- **name** (*str*) – e.g. name of identified progenitor.
- **concept** (*str*) – One of a 'formal UCD-like vocabulary of astronomical concepts', e.g. http://ivoat.ivoa.net/stars.supernova.Ia - see VOEvent spec.

voeventparse.misc.**EventIvorn**(*ivorn*, *cite_type*)
Used to cite earlier VOEvents.

Use in conjunction with *add_citations()*

Parameters

- **ivorn** (*str*) – It is assumed this will be copied verbatim from elsewhere, and so these should have any prefix (e.g. 'ivo://','http://') already in place - the function will not alter the value.
- **cite_type** (*definitions.cite_types*) – String conforming to one of the standard citation types.

voeventparse.misc.**Citation**(*ivorn*, *cite_type*)
Deprecated alias of *EventIvorn()*

1.4.3 voeventparse.convenience - Convenience routines

Convenience routines for common actions on VOEvent objects

voeventparse.convenience.**get_event_time_as_utc**(*voevent*, *index=0*)
Extracts the event time from a given *WhereWhen.ObsDataLocation*.

Returns a datetime (timezone-aware, UTC).

Accesses a *WhereWhere.ObsDataLocation.ObservationLocation* element and returns the AstroCoords.Time.TimeInstant.ISOTime element, converted to a (UTC-timezoned) datetime.

Note that a packet may include multiple 'ObsDataLocation' entries under the 'WhereWhen' section, for example giving locations of an object moving over time. Most packets will have only one, however, so the default is to access the first.

This function now implements conversion from the TDB (Barycentric Dynamical Time) time scale in ISOTime format, since this is the format used by GAIA VOEvents. (See also http://docs.astropy.org/en/stable/time/#time-scale)

Other timescales (i.e. TT, GPS) will presumably be formatted as a TimeOffset, parsing this format is not yet implemented.

Parameters

- **voevent** (*voeventparse.voevent.Voevent*) – Root node of the VOevent etree.
- **index** (*int*) – Index of the ObsDataLocation to extract an ISOtime from.

Returns Datetime representing the event-timestamp, converted to UTC (timezone aware).

Return type datetime.datetime

voeventparse.convenience.**get_event_position**(*voevent*, *index=0*)

Extracts the *AstroCoords* from a given *WhereWhen.ObsDataLocation*.

Note that a packet may include multiple 'ObsDataLocation' entries under the 'WhereWhen' section, for example giving locations of an object moving over time. Most packets will have only one, however, so the default is to just return co-ords extracted from the first.

Parameters

- **voevent** (*voeventparse.voevent.Voevent*) – Root node of the VOEvent etree.
- **index** (*int*) – Index of the ObsDataLocation to extract AstroCoords from.

Returns The sky position defined in the ObsDataLocation.

Return type Position (*Position2D*)

voeventparse.convenience.**get_grouped_params**(*voevent*)

Fetch grouped Params from the *What* section of a voevent as an omdict.

This fetches 'grouped' Params, i.e. those enclosed in a Group element, and returns them as a nested dict-like structure, keyed by GroupName->ParamName->AttribName.

Note that since multiple Params may share the same ParamName, the returned data-structure is actually an orderedmultidict.omdict and has extra methods such as 'getlist' to allow retrieval of all values.

Parameters **voevent** (*voeventparse.voevent.Voevent*) – Root node of the VOevent etree.

Returns (orderedmultidict.omdict): Mapping of `ParamName->Attribs`. Typical access like so:

```
foo_val = top_params['foo']['value']
# If there are multiple Param entries named 'foo':
all_foo_vals = [atts['value'] for atts in top_params.getlist('foo')]
```

voeventparse.convenience.**get_toplevel_params**(*voevent*)

Fetch ungrouped Params from the *What* section of a voevent as an omdict.

This fetches 'toplevel' Params, i.e. those not enclosed in a Group element, and returns them as a nested dict-like structure, keyed like ParamName->AttribName.

Note that since multiple Params may share the same ParamName, the returned data-structure is actually an orderedmultidict.omdict and has extra methods such as 'getlist' to allow retrieval of all values.

Any Params with no defined name (technically off-spec, but not invalidated by the XML schema) are returned under the dict-key `None`.

Parameters **voevent** (*voeventparse.voevent.Voevent*) – Root node of the VOevent etree.

Returns (orderedmultidict.omdict): Mapping of `ParamName->Attribs`. Typical access like so:

```
foo_val = top_params['foo']['value']
# If there are multiple Param entries named 'foo':
all_foo_vals = [atts['value'] for atts in top_params.getlist('foo')]
```

voeventparse.convenience.**pull_astro_coords**(*voevent*, *index=0*)

Deprecated alias of *get_event_position()*

voeventparse.convenience.**pull_isotime**(*voevent*, *index=0*)
Deprecated alias of *get_event_time_as_utc()*

voeventparse.convenience.**pull_params**(*voevent*)
Attempts to load the *What* section of a voevent as a nested dictionary.

> **Warning:** Deprecated due to *Missing name attributes* issues.
>
> *Param* or *Group* entries which are missing the *name* attribute will be entered under a dictionary key of `None`. This means that if there are multiple entries missing the *name* attribute then earlier entries will be overwritten by later entries, so you will not be able to use this convenience routine effectively. Use *get_grouped_params()* and *get_toplevel_params()* instead.

Parameters **voevent** (*voeventparse.voevent.Voevent*) – Root node of the VOevent etree.

Returns

Mapping of `Group->Param->Attribs`. Access like so:

```
foo_param_val = what_dict['GroupName']['ParamName']['value']
```

Note: Parameters without a group are indexed under the key 'None' - otherwise, we might get name-clashes between *params* and *groups* (unlikely but possible) so for ungrouped Params you'll need something like:

```
what_dict[None]['ParamName']['value']
```

Return type dict

voeventparse.convenience.**prettystr**(*subtree*)
Print an element tree with nice indentation.

Prettyprinting a whole VOEvent often doesn't seem to work, probably for issues relating to whitespace cf. http://lxml.de/FAQ.html#why-doesn-t-the-pretty-print-option-reformat-my-xml-output This function is a quick workaround for prettyprinting a subsection of a VOEvent, for easier desk-checking.

Parameters **subtree(** – class'lxml.etree.ElementTree'): A node in the VOEvent element tree.

Returns Prettyprinted string representation of the raw XML.

Return type str

1.4.4 voeventparse.definitions - Standard or common string values

This module simply serves to store the XML schema, a 'skeleton' VOEvent xml document for creation of new instances, and various other minor definitions.

These values may be used in place of literal strings, to allow autocompletion and document the fact that they are 'standardized' values.

class voeventparse.definitions.**roles**

observation = 'observation'

prediction = 'prediction'

utility = 'utility'

test = 'test'

class voeventparse.definitions.**sky_coord_system**
Common coordinate system identifiers. See also *Position2D*.

This is not a simple combinatorial mix of all components, it's a reproduction of the enumeration in the XML schema. Note some entries in the schema enumeration are repeated, which leads me to think it may be flawed, but that's an investigation for another day.

gps_fk5_topo = 'GPS-FK5-TOPO'

gps_icrs_geo = 'GPS-ICRS-GEO'

gps_icrs_topo = 'GPS-ICRS-TOPO'

tdb_fk5_bary = 'TDB-FK5-BARY'

tdb_icrs_bary = 'TDB-ICRS-BARY'

tt_fk5_geo = 'TT-FK5-GEO'

tt_fk5_topo = 'TT-FK5-TOPO'

tt_icrs_geo = 'TT-ICRS-GEO'

tt_icrs_topo = 'TT-ICRS-TOPO'

utc_fk5_geo = 'UTC-FK5-GEO'

utc_fk5_topo = 'UTC-FK5-TOPO'

utc_icrs_geo = 'UTC-ICRS-GEO'

utc_icrs_topo = 'UTC-ICRS-TOPO'

class voeventparse.definitions.**observatory_location**
Common generic values for the WhereWhen.ObservatoryLocation attribute.

geosurface = 'GEOSURFACE'

geolunar = 'GEOLUN'

class voeventparse.definitions.**units**
Unit abbreviations as defined by CDS (incomplete listing)

cf http://vizier.u-strasbg.fr/doc/catstd-3.2.htx

degree = 'deg'

degrees = 'deg'

arcsecond = 'arcsec'

milliarcsecond = 'mas'

day = 'd'

hour = 'h'

minute = 'min'

year = 'yr'

count = 'ct'

hertz = 'Hz'

jansky = 'Jy'

magnitude = 'mag'

class voeventparse.definitions.**cite_types**
Possible types of *EventIvorn()*

followup = 'followup'

supersedes = 'supersedes'

retraction = 'retraction'

1.5 Change history

1.5.1 1.0.2 - 2018/02/10

Fixes

Fix some minor issues with the the 'prettystr' routine:

- Ensure return of a string type in Python3 (was bytes).
- Fix printing of whole packet (#6, with thanks to @henrilouvin).

1.5.2 1.0.1 - 2016/11/28

Fixes

Minor bugfix - bring back importing of deprecated 'pull_astro_coords' function into the package-root namespace.

1.5.3 1.0.0 - 2016/11/28

API Changes

Some routines have been renamed, with the old aliases preserved for backwards compatibility for now:

- `add_where_when` now has an extra boolean parameter, 'allow_tz_naive_datetime' which defaults to False - so by default you must supply a **timezone-aware** datetime (this should help to avoid mistakenly supplying a timezone-naive datetime which is non-UTC).
- `Citation` is now deprecated in favour of the alias `EventIvorn`.
- `pull_isotime` is now deprecated in favour of the alias `get_event_time_as_utc`.
- `pull_astro_coords` is now deprecated in favour of the alias `get_event_position`.
- `pull_params` is now deprecated in favour of the improved replacement functions `get_grouped_params` and `get_toplevel_params`. Separating this functionality into two routines ensures return of datastructures with sensible nesting-depth (i.e. `toplevel[ParamName][AttrName]` or `grouped[GroupName][ParamName][AttrName]`), and avoids problems with GroupNames clashing with top-level ParamNames, etc. The returned datastructures are of type orderedmultidict.omdict, which provides robust handling of duplicated names.

Docs

Documentation now includes tutorial material which was previously hosted in a separate GitHub repo.

Fixes

Fix a regression from 0.9.8: Switching from from iso8601 library to astropy for ISO-format timestamp parsing introduced a failure case, as astropy does not parse timestamps with the '+0' timezone-signifier suffix. Timestamps of this style are now parsed correctly. In addition, `add_where_when` will now generate ISO-format timestamps that do not include the suffix (since strictly speaking the VOEvent standard specifies the UTC timezone - or something altogether non-terrestial, e.g. TDB - already).

General refactoring

Library code now resides under /src, which provides more reliable testing cf https://blog.ionelmc.ro/2014/05/25/python-packaging/, https://hynek.me/articles/testing-packaging/.

1.5.4 0.9.8 - 2016/11/09

Enhancement to `pull_isotime` convenience function: Add support for conversion of event-timestamps to UTC from the TDB (Barycentric Dynamical Time) format. This is now in use 'in the wild' for the GAIA VOEvent stream. (Support for remaining VOEvent timescales 'GPS' and 'TT' has been considered but needs a motivating use-case, see https://github.com/timstaley/voevent-parse/issues/5) NB this functionality introduces a dependence on Astropy>=1.2. (While this seems a little like using a sledgehammer to crack a nut, it's likely to be a useful library for anyone handling VOEvents anyway, and barycentric time correction codes are hard to find).

1.5.5 0.9.7 - 2016/10/31

Identical to 0.96, fixes a packaging issue most likely due to temporary files / dev installation confusing the packager.

1.5.6 0.9.6 - 2016/10/31

Minor bugfix to `pull_params`: Don't bork on missing Param name. However, note that if multiple Params are present without a *name* attribute then this convenience routine doesn't really make sense - see warning in docs.

1.5.7 0.9.5 - 2016/05/03

Switch to versioneer for version numbering / release tagging. CF https://github.com/warner/python-versioneer

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

V

Index

P

R

S

T

U

V

Y