pvl Documentation

Release 1.3.2

William Trevor Olson

Aug 11, 2022

Contents:

CHAPTER 1

pvl

Python implementation of a PVL (Parameter Value Language) library.

- Free software: BSD license
- Documentation: http://pvl.readthedocs.org.
- Support for Python 3.6 and higher (avaiable via pypi and conda).
- PlanetaryPy Affiliate Package.

PVL is a markup language, like JSON or YAML, commonly employed for entries in the Planetary Data System used by NASA to archive mission data, among other uses. This package supports both encoding and decoding a variety of PVL 'flavors' including PVL itself, ODL, NASA PDS 3 Labels, and USGS ISIS Cube Labels.

1.1 Installation

Can either install with pip or with conda.

To install with pip, at the command line:

```
$ pip install pvl
```

Directions for installing with conda-forge:

Installing `pvl` from the conda-forge channel can be achieved by adding conda-forge to your channels with:

```
conda config --add channels conda-forge
```

Once the conda-forge channel has been enabled, `pvl` can be installed with:

```
conda install pvl
```

It is possible to list all of the versions of `pvl` available on your platform with:

```
conda search pvl --channel conda-forge
```

1.2 Basic Usage

`pvl` exposes an API familiar to users of the standard library `json` module.

Decoding is primarily done through `pvl.load()` for file-like objects and `pvl.loads()` for strings:

```
>>> import pvl
>>> module = pvl.loads("""
...     foo = bar
...     items = (1, 2, 3)
...     END
... """)
>>> print(module)
PVLModule([
  ('foo', 'bar')
  ('items', [1, 2, 3])
])
>>> print(module['foo'])
bar
```

There is also a `pvl.loadu()` to which you can provide the URL of a file that you would normally provide to `pvl.load()`.

You may also use `pvl.load()` to read PVL text directly from an image that begins with PVL text:

```
>>> import pvl
>>> label = pvl.load('tests/data/pattern.cub')
>>> print(label)
PVLModule([
  ('IsisCube',
   {'Core': {'Dimensions': {'Bands': 1,
                            'Lines': 90,
                            'Samples': 90},
             'Format': 'Tile',
             'Pixels': {'Base': 0.0,
                        'ByteOrder': 'Lsb',
                        'Multiplier': 1.0,
                        'Type': 'Real'},
             'StartByte': 65537,
             'TileLines': 128,
             'TileSamples': 128}})
  ('Label', PVLObject([
```

(continues on next page)

(continued from previous page)

```
    ('Bytes', 65536)
  ]))
])
>>> print(label['IsisCube']['Core']['StartByte'])
65537
```

Similarly, encoding Python objects as PVL text is done through `pvl.dump()` and `pvl.dumps()`:

```
>>> import pvl
>>> print(pvl.dumps({
...     'foo': 'bar',
...     'items': [1, 2, 3]
... }))
FOO   = bar
ITEMS = (1, 2, 3)
END
```

`pvl.PVLModule` objects may also be pragmatically built up to control the order of parameters as well as duplicate keys:

```
>>> import pvl
>>> module = pvl.PVLModule({'foo': 'bar'})
>>> module.append('items', [1, 2, 3])
>>> print(pvl.dumps(module))
FOO   = bar
ITEMS = (1, 2, 3)
END
```

A `pvl.PVLModule` is a `dict`-like container that preserves ordering as well as allows multiple values for the same key. It provides similar semantics to a `list` of key/value `tuples` but with `dict`-style access:

```
>>> import pvl
>>> module = pvl.PVLModule([
...     ('foo', 'bar'),
...     ('items', [1, 2, 3]),
...     ('foo', 'remember me?'),
... ])
>>> print(module['foo'])
bar
>>> print(module.getall('foo'))
['bar', 'remember me?']
>>> print(module.items())
ItemsView(PVLModule([
  ('foo', 'bar')
  ('items', [1, 2, 3])
  ('foo', 'remember me?')
]))
>>> print(pvl.dumps(module))
FOO   = bar
ITEMS = (1, 2, 3)
FOO   = 'remember me?'
END
```

However, there are some aspects to the default `pvl.PVLModule` that are not entirely aligned with the modern Python 3 expectations of a Mapping object. If you would like to experiment with a more Python-3-ic object, you could instantiate a `pvl.collections.PVLMultiDict` object, or `import pvl.new as pvl` in your code to have the loaders return objects of this type (and then easily switch back by just changing the import statement).

To learn more about how PVLMultiDict is different from the existing OrderedMultiDict that PVLModule is derived from, please read the new PVLMultiDict documentation.

The intent is for the loaders (`pvl.load()`, `pvl.loads()`, and `pvl.loadu()`) to be permissive, and attempt to parse as wide a variety of PVL text as possible, including some kinds of 'broken' PVL text.

On the flip side, when dumping a Python object to PVL text (via `pvl.dumps()` and `pvl.dump()`), the library will default to writing PDS3-Standards-compliant PVL text, which in some ways is the most restrictive, but the most likely version of PVL text that you need if you're writing it out (this is different from pre-1.0 versions of `pvl`).

You can change this behavior by giving different parameters to the loaders and dumpers that define the grammar of the PVL text that you're interested in, as well as custom parsers, decoders, and encoders.

For more information on custom serilization and deseralization see the full documentation.

1.3 Contributing

Feedback, issues, and contributions are always gratefully welcomed. See the contributing guide for details on how to help and setup a development environment.

CHAPTER 2

Parsing PVL text

Table of Contents

2.1 From a File

The *pvl.load()* function parses PVL text from a file or stream and returns a `dict`-like object (*pvl.PVLModule* by default) containing information from that text. This documentation will explain how to use the module as well as some sample code to use the module efficiently.

2.1.1 Simple Use

How to use *pvl.load()* to get a single value:

```
>>> from pathlib import Path
>>> import pvl
>>> path = Path('tests/data/pds3/simple_image_1.lbl')
```

(continues on next page)

(continued from previous page)

```
>>> pvl.load(path)['RECORD_TYPE']
'FIXED_LENGTH'

>>> import pvl
>>> img = 'tests/data/pds3/simple_image_1.lbl'
>>> pvl.load(img)['RECORD_TYPE']
'FIXED_LENGTH'

>>> import pvl
>>> img = 'tests/data/pds3/simple_image_1.lbl'
>>> with open(img, 'r+') as r:
...     print(pvl.load(r)['RECORD_TYPE'])
FIXED_LENGTH
```

2.1.2 Detailed Use

To view the image label of an ISIS cube as a dictionary:

```
>>> import pvl
>>> img = 'tests/data/pattern.cub'
>>> module = pvl.load(img)
>>> print(module)
PVLModule([
  ('IsisCube',
   {'Core': {'Dimensions': {'Bands': 1,
                            'Lines': 90,
                            'Samples': 90},
             'Format': 'Tile',
             'Pixels': {'Base': 0.0,
                        'ByteOrder': 'Lsb',
                        'Multiplier': 1.0,
                        'Type': 'Real'},
             'StartByte': 65537,
             'TileLines': 128,
             'TileSamples': 128}})
  ('Label', PVLObject([
    ('Bytes', 65536)
  ]))
])
```

Not all image labels are formatted the same so different labels will have different information that you can obtain. To view what information you can extract use the `.keys()` function:

```
>>> import pvl
>>> img = 'tests/data/pds3/simple_image_1.lbl'
>>> lbl = pvl.load(img)
>>> lbl.keys()
KeysView(['PDS_VERSION_ID', 'RECORD_TYPE', 'RECORD_BYTES', 'LABEL_RECORDS', 'FILE_
↪RECORDS', '^IMAGE', 'IMAGE'])
```

... now you can just copy and paste from this list:

```
>>> lbl['RECORD_TYPE']
'FIXED_LENGTH'
```

The list `.keys()` returns is out of order, to see the keys in the order of the dictionary use `.items()` function:

```
>>> import pvl
>>> img = 'tests/data/pds3/simple_image_1.lbl'
>>> for item in pvl.load(img).items():
...     print(item[0])
PDS_VERSION_ID
RECORD_TYPE
RECORD_BYTES
LABEL_RECORDS
FILE_RECORDS
^IMAGE
IMAGE
```

We can take advantage of the fact `.items()` returns a list in order and use the index number of the key instead of copying and pasting. This will make extracting more than one piece of information at time more convenient. For example, if you want to print out the first 5 pieces of information:

```
>>> import pvl
>>> img = 'tests/data/pds3/simple_image_1.lbl'
>>> pvl_items = pvl.load(img).items()
>>> for n in range(0, 5):
...     print(pvl_items[n][0], pvl_items[n][1])
PDS_VERSION_ID PDS3
RECORD_TYPE FIXED_LENGTH
RECORD_BYTES 824
LABEL_RECORDS 1
FILE_RECORDS 601
```

... some values have sub-dictionaries. You can access those by:

```
>>> print(pvl.load(img)['IMAGE'].keys())
KeysView(['LINES', 'LINE_SAMPLES', 'SAMPLE_TYPE', 'SAMPLE_BITS', 'MEAN', 'MEDIAN',
↪'MINIMUM', 'MAXIMUM', 'STANDARD_DEVIATION', 'CHECKSUM'])
>>> print(pvl.load(img)['IMAGE']['SAMPLE_BITS'])
8
```

Another way of using *pvl.load()* is to use Python's `with open()` command. Otherwise using this method is very similar to using the methods described above:

```
>>> import pvl
>>> with open('tests/data/pattern.cub','r') as r:
...     print(pvl.load(r)['Label']['Bytes'])
65536
```

2.2 From a String

The *pvl.loads()* function returns a Python object (typically a *pvl.PVLModule* object which is `dict`-like) based on parsing the PVL text in the string parameter that it is given.

2.2.1 Simple Use

How to use *pvl.loads()*:

```
>>> import pvl
>>> s = """String = 'containing the label of the image'
... key = value
... END
... """
>>> pvl.loads(s).keys()
KeysView(['String', 'key'])

>>> pvl.loads(s)['key']
'value'
```

2.2.2 Detailed Use

To view the image label dictionary:

```
>>> import pvl
>>> string = """Object = IsisCube
...   Object = Core
...     StartByte   = 65537
...     Format      = Tile
...     TileSamples = 128
...     TileLines   = 128
...
...   End_Object
... End_Object
...
... Object = Label
...   Bytes = 65536
... End_Object
... End"""
>>> print(pvl.loads(string))
PVLModule([
  ('IsisCube',
   {'Core': {'Format': 'Tile',
             'StartByte': 65537,
             'TileLines': 128,
             'TileSamples': 128}})
  ('Label', PVLObject([
    ('Bytes', 65536)
  ]))
])
```

. . . to view the keys available:

```
>>> print(pvl.loads(string).keys())
KeysView(['IsisCube', 'Label'])
```

. . . and to see the information contained in the keys:

```
>>> print(pvl.loads(string)['Label'])
PVLObject([
  ('Bytes', 65536)
])
```

. . . and what is in the sub-dictionary:

```
>>> print(pvl.loads(string)['Label']['Bytes'])
65536
```

By default, *pvl.loads()* and *pvl.load()* are very permissive, and do their best to attempt to parse a wide variety of PVL 'flavors.'

If a parsed label has a parameter with a missing value, the default behavior of these functions will be to assign a *pvl.parser.EmptyValueAtLine* object as the value:

```
>>> string = """
... Object = Label
...   A =
... End_Object
... End"""

>>> print(pvl.loads(string))
PVLModule([
  ('Label',
   {'A': EmptyValueAtLine(3 does not have a value. Treat as an empty string)})
])
```

Stricter parsing can be accomplished by passing a different grammar object (e.g. *pvl.grammar.PVLGrammar*, *pvl.grammar.ODLGrammar*) to *pvl.loads()* or *pvl.load()*:

```
>>> import pvl
>>> some_pvl = """Comments = "PVL and ODL only allow /* */ comments"
... /* like this */
... # but people use hash-comments all the time
... END
... """
>>> print(pvl.loads(some_pvl))
PVLModule([
  ('Comments', 'PVL and ODL only allow /* */ comments')
])
>>> pvl.loads(some_pvl, grammar=pvl.grammar.PVLGrammar())
Traceback (most recent call last):
  ...
pvl.exceptions.LexerError: (LexerError(...), 'Expecting an Aggregation Block, an␣
↪Assignment Statement, or an End Statement, but found "#" : line 3 column 1 (char␣
↪67) near "like this */\n# but people"')
```

2.3 From a URL

The `pvl.loadu()` function returns a Python object (typically a *pvl.PVLModule* object which is `dict`-like) based on parsing the PVL text in the data returned from a URL.

This is very similar to parsing PVL text from a file, but you use `pvl.loadu()` instead:

```
>>> import pvl
>>> url = 'https://hirise-pds.lpl.arizona.edu/PDS/RDR/ESP/ORB_017100_017199/ESP_
↪017173_1715/ESP_017173_1715_RED.LBL'
>>> pvl.loadu(url)['VIEWING_PARAMETERS']['PHASE_ANGLE']
Quantity(value=50.784875, units='DEG')
```

Of course, other kinds of URLs, like file, ftp, rsync, sftp and more can be used.

2.4 Return non-standard objects

The "loaders" return a dict-like filled with Python objects based on the types inferred from the PVL-text. Sometimes you may want the *pvl* library to return different types in the dict-like, and *pvl* has some limited capacity for that (so far just real and quantity types).

Normally real number values in the PVL-text will be returned as Python `float` objects. However, what if you wanted all of the real values to be returned in the dict-like as Python `decimal.Decimal` objects (because you wanted to preserve numeric precision)? You can do that by providing the object type you want via the `real_cls` argument of a decoder constructor, like so:

```
>>> from decimal import Decimal
>>> import pvl
>>> text = "gigawatts = 1.210"
>>>
>>> flo = pvl.loads(text)
>>> print(flo)
PVLModule([
  ('gigawatts', 1.21)
])
>>>
>>> print(type(flo["gigawatts"]))
<class 'float'>
>>> dec = pvl.loads(text, decoder=pvl.decoder.OmniDecoder(real_cls=Decimal))
>>> print(dec)
PVLModule([
  ('gigawatts', Decimal('1.210'))
])
>>> print(type(dec["gigawatts"]))
<class 'decimal.Decimal'>
```

Any class that can be passed a `str` object to initialize an object can be provided to `real_cls`, but it should emit a `ValueError` if it is given a string that should not be converted to a real number value.

To learn more about quantity classes in *pvl*, please see *Quantities: Values and Units*.

CHAPTER 3

Writing out PVL text

This documentation explains how you can use `pvl.dump()` and `pvl.dumps()` so you can change, add, and/or write out a Python `dict`-like object as PVL text either to a `str` or a file. This documentation assumes that you've read about how to parse PVL text and know how `pvl.load()` and `pvl.loads()` work.

The examples primarily use an ISIS Cube image label format, which typically doesn't conform to PDS 3 standards, so pay attention to the differences between the PVL text that is loaded, versus the PDS 3-compliant PVL text that is dumped.

However, this library can write/alter any PVL compliant label.

Table of Contents

3.1 Writing PVL Text to a File

The `pvl.dump()` function allows you to write out a `dict`-like Python object (typically a `pvl.PVLModule` object) to a file as PVL text.

3.1.1 Simple Use

Read a label from a file:

```
>>> import pvl
>>> pvl_file = 'tests/data/pds3/tiny1.lbl'
>>> label = pvl.load(pvl_file)
>>> print(label)
PVLModule([
  ('PDS_VERSION_ID', 'PDS3')
])
```

. . . then you can change a value:

```
>>> label['PDS_VERSION_ID'] = 42
>>> print(label)
PVLModule([
  ('PDS_VERSION_ID', 42)
])
```

. . . then add keys to the `label` object:

```
>>> label['New_Key'] = 'New_Value'
>>> print(label)
PVLModule([
  ('PDS_VERSION_ID', 42)
  ('New_Key', 'New_Value')
])
```

. . . and then write out the PVL text to a file:

```
>>> pvl.dump(label, 'new.lbl')
54
```

`pvl.dump()` returns the number of characters written to the file.

3.1.2 Changing A Key

More complicated parameter value change.

Load some PVL text from a file:

```
>>> import pvl
>>> img = 'tests/data/pattern.cub'
>>> label = pvl.load(img)
>>> print(label['IsisCube']['Core']['Format'])
Tile
```

. . . then change key 'Format' to 'Changed_Value':

```
>>> label['IsisCube']['Core']['Format'] = 'Changed_Value'
```

. . . then writing out file with new value:

```
>>> new_file = 'new.lbl'
>>> pvl.dump(label, new_file)
494
```

If you then try to show the changed value in the file, you'll get an error:

```
>>> new_label = pvl.load(new_file)
>>> print(new_label['IsisCube']['Core']['Format'])
Traceback (most recent call last):
   ...
KeyError: 'Format'
```

This is because the default for *pvl.dump()* and *pvl.dumps()* is to write out PDS3-Standards-compliant PVL, in which the parameter values (but not the aggregation block names) are uppercased:

```
>>> print(new_label['IsisCube']['Core'].keys())
KeysView(['STARTBYTE', 'FORMAT', 'TILESAMPLES', 'TILELINES', 'Dimensions', 'Pixels'])
>>> print(new_label['IsisCube']['Core']['FORMAT'])
Changed_Value
```

Clean up:

```
>>> import os
>>> os.remove(new_file)
```

Yes, this case difference is weird, yes, this means that you need to be aware of the case of different keys in your *pvl.PVLModule* objects.

3.2 Writing PVL Text to a String

The *pvl.dumps()* function allows you to convert a `dict`-like Python object (typically a *pvl.PVLModule* object) to a Python `str` object which contains the PVL text.

3.2.1 Simple Use

Get started, as above:

```
>>> import pvl
>>> pvl_file = 'tests/data/pds3/tiny1.lbl'
>>> label = pvl.load(pvl_file)
>>> print(label)
PVLModule([
  ('PDS_VERSION_ID', 'PDS3')
])
```

. . . then change a value, and add keys:

```
>>> label['PDS_VERSION_ID'] = 42
>>> label['New_Param'] = 'New_Value'
>>> print(label)
PVLModule([
  ('PDS_VERSION_ID', 42)
  ('New_Param', 'New_Value')
])
```

. . . then write to a string:

```
>>> print(pvl.dumps(label))
PDS_VERSION_ID = 42
NEW_PARAM      = New_Value
END
```

Here we can see the effects of the PDS3LabelEncoder in the default behavior of *pvl.dumps()*: it uppercases the parameters, and puts a blank line after the END statement. If we were to use the PVLEncoder, you can see different behavior:

```
>>> print(pvl.dumps(label, encoder=pvl.encoder.PVLEncoder()))
PDS_VERSION_ID = 42;
New_Param      = New_Value;
END;
```

3.2.2 Adding A Key

More complicated:

```
>>> import pvl
>>> pvl_file = 'tests/data/pds3/group1.lbl'
>>> label = pvl.load(pvl_file)
>>> print(label)
PVLModule([
  ('PDS_VERSION_ID', 'PDS3')
  ('IMAGE',
   {'CHECKSUM': 25549531,
    'MAXIMUM': 255,
    'STANDARD_DEVIATION': 16.97019})
  ('SHUTTER_TIMES', PVLGroup([
    ('START', 1234567)
    ('STOP', 2123232)
  ]))
])
```

. . . then add a new key and value to a sub group:

```
>>> label['New_Key'] = 'New_Value'
>>> label['IMAGE']['New_SubKey'] = 'New_SubValue'
>>> print(label)
PVLModule([
  ('PDS_VERSION_ID', 'PDS3')
  ('IMAGE',
   {'CHECKSUM': 25549531,
    'MAXIMUM': 255,
    'New_SubKey': 'New_SubValue',
    'STANDARD_DEVIATION': 16.97019})
  ('SHUTTER_TIMES', PVLGroup([
    ('START', 1234567)
    ('STOP', 2123232)
  ]))
  ('New_Key', 'New_Value')
])
```

. . . then when we dump, the default is to write PDS3 Labels, so the parameters are uppercased:

```
>>> print(pvl.dumps(label))
PDS_VERSION_ID = PDS3
OBJECT = IMAGE
  MAXIMUM            = 255
  STANDARD_DEVIATION = 16.97019
  CHECKSUM           = 25549531
  NEW_SUBKEY         = New_SubValue
END_OBJECT = IMAGE
GROUP = SHUTTER_TIMES
  START = 1234567
  STOP  = 2123232
END_GROUP = SHUTTER_TIMES
NEW_KEY        = New_Value
END
```

3.2.3 Example with an ISIS cube file

```
>>> import pvl
>>> img = 'tests/data/pattern.cub'
>>> label = pvl.load(img)
>>> label['New_Key'] = 'New_Value'
>>> label_string = pvl.dumps(label)
>>> print(label_string)
OBJECT = IsisCube
  OBJECT = Core
    STARTBYTE   = 65537
    FORMAT      = Tile
    TILESAMPLES = 128
    TILELINES   = 128
    GROUP = Dimensions
      SAMPLES = 90
      LINES   = 90
      BANDS   = 1
    END_GROUP = Dimensions
    GROUP = Pixels
      TYPE       = Real
      BYTEORDER  = Lsb
      BASE       = 0.0
      MULTIPLIER = 1.0
    END_GROUP = Pixels
  END_OBJECT = Core
END_OBJECT = IsisCube
OBJECT = Label
  BYTES = 65536
END_OBJECT = Label
NEW_KEY      = New_Value
END
```

3.2.4 PVL text for ISIS program consumption

There are a number of ISIS programs that take PVL text files as a way of allowing users to provide more detailed inputs. To write PVL text that is readable by ISIS, you can use the *pvl.encoder.ISISEncoder*. Here's an example of creating a map file used by the ISIS program `cam2map`. Since `cam2map` needs the 'Mapping' aggregation to be a PVL Group, you must use the *pvl.PVLGroup* object to assign to 'Mapping' rather than just a dict-like (which gets

encoded as a PVL Object by default). You'd normally use *pvl.dump()* to write to a file, but we use *pvl.dumps()* here to show what you'd get:

```
>>> import pvl
>>> subsc_lat = 10
>>> subsc_lon = 10
>>> map_pvl = {'Mapping': pvl.PVLGroup({'ProjectionName': 'Orthographic',
...                                     'CenterLatitude': subsc_lat,
...                                     'CenterLongitude': subsc_lon})}
>>> print(pvl.dumps(map_pvl, encoder=pvl.encoder.ISISEncoder()))
Group = Mapping
  ProjectionName  = Orthographic
  CenterLatitude  = 10
  CenterLongitude = 10
End_Group = Mapping
END
```

3.2.5 Pre-1.0 pvl dump behavior

If you don't like the new default behavior of writing out PDS3 Label Compliant PVL text, then just using an encoder with some different settings will get you the old style:

```
>>> import pvl
>>> img = 'tests/data/pattern.cub'
>>> label = pvl.load(img)
>>> print(pvl.dumps(label))
OBJECT = IsisCube
  OBJECT = Core
    STARTBYTE   = 65537
    FORMAT      = Tile
    TILESAMPLES = 128
    TILELINES   = 128
    GROUP = Dimensions
      SAMPLES = 90
      LINES   = 90
      BANDS   = 1
    END_GROUP = Dimensions
    GROUP = Pixels
      TYPE       = Real
      BYTEORDER  = Lsb
      BASE       = 0.0
      MULTIPLIER = 1.0
    END_GROUP = Pixels
  END_OBJECT = Core
END_OBJECT = IsisCube
OBJECT = Label
  BYTES = 65536
END_OBJECT = Label
END

>>> print(pvl.dumps(label, encoder=pvl.PVLEncoder(end_delimiter=False)))
...
BEGIN_OBJECT = IsisCube
  BEGIN_OBJECT = Core
    StartByte   = 65537
    Format      = Tile
```

(continues on next page)

(continued from previous page)

```
    TileSamples = 128
    TileLines   = 128
    BEGIN_GROUP = Dimensions
      Samples = 90
      Lines   = 90
      Bands   = 1
    END_GROUP = Dimensions
    BEGIN_GROUP = Pixels
      Type       = Real
      ByteOrder  = Lsb
      Base       = 0.0
      Multiplier = 1.0
    END_GROUP = Pixels
  END_OBJECT = Core
END_OBJECT = IsisCube
BEGIN_OBJECT = Label
  Bytes = 65536
END_OBJECT = Label
END
```

... of course, to really get the true old behavior, you should also use the carriage return/newline combination line endings, and encode the string as a bytearray, since that is the Python type that the pre-1.0 library produced:

```
>>> print(pvl.dumps(label, encoder=pvl.PVLEncoder(end_delimiter=False,
...                                               newline='\r\n')).encode())
b'BEGIN_OBJECT = IsisCube\r\n  BEGIN_OBJECT = Core\r\n    StartByte   = 65537\r\n
↪Format      = Tile\r\n    TileSamples = 128\r\n    TileLines   = 128\r\n    BEGIN_
↪GROUP = Dimensions\r\n      Samples = 90\r\n      Lines   = 90\r\n      Bands   =
↪1\r\n    END_GROUP = Dimensions\r\n    BEGIN_GROUP = Pixels\r\n      Type       =
↪Real\r\n      ByteOrder  = Lsb\r\n      Base       = 0.0\r\n      Multiplier = 1.
↪0\r\n    END_GROUP = Pixels\r\n  END_OBJECT = Core\r\nEND_OBJECT =
↪IsisCube\r\nBEGIN_OBJECT = Label\r\n  Bytes = 65536\r\nEND_OBJECT = Label\r\nEND'
```

CHAPTER 4

Quantities: Values and Units

The PVL specifications supports the notion that PVL Value Expressions can contain an optional PVL Units Expression that follows the PVL Value. This combination of information: a value followed by a unit can be represented by a single object that we might call a quantity.

There is no fundamental Python object type that represents a value and the units of that value. However, libraries like `astropy` and `pint` have implemented "quantity" objects (and managed to name them both Quantity, but they have slightly different interfaces). In order to avoid optional dependencies, the `pvl` library provides the *`pvl.collections.Quantity`* class, implemented as a `collections.namedtuple` with a `value` and a `unit` parameter. However, the `unit` parameter is just a string and so the `pvl` quantity objects doesn't have the super-powers that the `astropy` and `pint` quntity objects do.

By default, this means that when PVL text is parsed by *`pvl.load()`* or *`pvl.loads()`* and when a PVL Value followed by a PVL Units Expression is encountered, a *`pvl.collections.Quantity`* object will be placed in the returned dict-like.

Likewise when *`pvl.dump()`* or *`pvl.dumps()`* encounters a *`pvl.collections.Quantity`* its value and units will be serialized with the right PVL syntax.

However, the `pvl` library also supports the use of other quantity objects.

4.1 Getting other quantity objects from PVL text

In order to get the parsing side of the `pvl` library to return a particular kind of quantity object when a PVL Value followed by a PVL Units Expression is found, you must pass the name of that quantity class to the decoder's `quantity_cls` argument. This quantity class's constructor must take two arguments, where the first will receive the PVL Value (as whatever Python type `pvl` determines it to be) and the second will receive the PVL Units Expression (as a `str`).

Examples of how to do this with *`pvl.load()`* or *`pvl.loads()`* are below for `astropy` and `pint`.

Depending on the PVL text that you are parsing, and the quantity class that you are using, you may get errors if the quantity class can't accept the PVL Units Expression, or if the *value* part of the quantity class can't handle all of the possible types of PVL Values (which can be Simple Values, Sets, or Sequences).

4.2 Writing out other quantity objects to PVL text

In order to get the encoding side of the `pvl` library to write out the correct kind of PVL text based on some quantity object is more difficult due to the wide variety of ways that quantity objects are written in 3rd party libaries. At this time, the `pvl` library can properly encode `pvl.collecitons.Quantity`, `astropy.units.Quantity`, and `pint.Quantity` objects (or objects that pass an `isinstance()` test for those objects). Any other kind of quantity object in the data structure passed to *`pvl.dump()`* or *`pvl.dumps()`* will just be encoded as a string.

Other types are possible, but require additions to the encoder in use. The `astropy.units.Quantity` object is already handled by the `pvl` library, but if it wasn't, this is how you would enable it. You just need the class name, the name of the property on the class that yields the value or magnitude (for `astropy.units.Quantity` that is `value`), and the property that yields the units (for `astropy.units.Quantity` that is `unit`). With those pieces in hand, we just need to instantiate an encoder and add the new quantity class and the names of those properties to it, and then pass it to *`pvl.dump()`* or *`pvl.dumps()`* as follows:

```
>>> import pvl
>>> from astropy import units as u
>>> my_label = dict(length=u.Quantity(15, u.m), velocity=u.Quantity(0.5, u.m / u.s))
>>> my_encoder = pvl.PDSLabelEncoder()
>>> my_encoder.add_quantity_cls(u.Quantity, 'value', 'unit')
>>> print(pvl.dumps(my_label, encoder=my_encoder))
LENGTH   = 15.0 <m>
VELOCITY = 0.5 <m / s>
END
```

4.3 astropy.units.Quantity

The Astropy Project has classes for handing Units and Quantities.

The `astropy.units.Quantity` object can be returned in the data structure returned from *`pvl.load()`* or *`pvl.loads()`*. Here is an example:

```
>>> import pvl
>>> pvl_text = "length = 42 <m/s>"
>>> regular = pvl.loads(pvl_text)
>>> print(regular['length'])
Quantity(value=42, units='m/s')
>>> print(type(regular['length']))
<class 'pvl.collections.Quantity'>

>>> from pvl.decoder import OmniDecoder
>>> from astropy import units as u
>>> w_astropy = pvl.loads(pvl_text, decoder=OmniDecoder(quantity_cls=u.Quantity))
>>> print(w_astropy)
PVLModule([
  ('length', <Quantity 42. m / s>)
])
>>> print(type(w_astropy['length']))
<class 'astropy.units.quantity.Quantity'>
```

However, in our example file and in other files you may parse, the units may be in upper case (e.g. KM, M), and by default, astropy will not recognize the name of these units. It will raise a handy exception, which, in turn, will be raised as a `pvl.parser.QuantityError` that will look like this:

```
pvl.parser.QuantityError: 'KM' did not parse as unit: At col
0, KM is not a valid unit. Did you mean klm or km? If this is
meant to be a custom unit, define it with 'u.def_unit'. To have
it recognized inside a file reader or other code, enable it
with 'u.add_enabled_units'. For details, see
http://docs.astropy.org/en/latest/units/combining_and_defining.html
```

So, in order to parse our file, do this:

```
>>> import pvl
>>> from pvl.decoder import OmniDecoder
>>> from astropy import units as u
>>> pvl_file = 'tests/data/pds3/units1.lbl'
>>> km_upper = u.def_unit('KM', u.km)
>>> m_upper = u.def_unit('M', u.m)
>>> u.add_enabled_units([km_upper, m_upper])
<astropy.units.core._UnitContext object at ...
>>> label = pvl.load(pvl_file, decoder=OmniDecoder(quantity_cls=u.Quantity))
>>> print(label)
PVLModule([
  ('PDS_VERSION_ID', 'PDS3')
  ('MSL:COMMENT', 'THING TEST')
  ('FLOAT_UNIT', <Quantity 0.414 KM>)
  ('INT_UNIT', <Quantity 4. M>)
])
>>> print(type(label['FLOAT_UNIT']))
<class 'astropy.units.quantity.Quantity'>
```

Similarly, `astropy.units.Quantity` objects can be encoded to PVL text by *`pvl.dump()`* or *`pvl.dumps()`* without any particular special handling. Here is an example:

```
>>> import pvl
>>> from astropy import units as u
>>> my_label = dict(length=u.Quantity(15, u.m), velocity=u.Quantity(0.5, u.m / u.s))
>>> print(pvl.dumps(my_label))
LENGTH   = 15.0 <m>
VELOCITY = 0.5 <m / s>
END
```

4.4 pint.Quantity

The Pint library also deals with quantities.

The `pint.Quantity` object can also be returned in the data structure returned from *`pvl.load()`* or *`pvl.loads()`* if you would prefer to use those objects. Here is an example:

```
>>> import pvl
>>> pvl_text = "length = 42 <m/s>"
>>> from pvl.decoder import OmniDecoder
>>> import pint
>>> w_pint = pvl.loads(pvl_text, decoder=OmniDecoder(quantity_cls=pint.Quantity))
>>> print(w_pint)
PVLModule([
  ('length', <Quantity(42, 'meter / second')>)
])
```

(continues on next page)

(continued from previous page)

```
>>> print(type(w_pint['length']))
<class 'pint.quantity.Quantity'>
```

Just as with `astropy.units.Quantity`, `pint.Quantity` doesn't recognize the upper case units, and will raise an error like this:

```
pint.errors.UndefinedUnitError: 'KM' is not defined in the unit registry
```

So, in order to parse our file with uppercase units, you can create a units definition file to add aliases and units to the pint 'registry'. When doing this programmatically note that if you define a registry on-the-fly, you must use the registry's Quantity to the `quantity_cls` argument:

```
>>> import pvl
>>> from pvl.decoder import OmniDecoder
>>> import pint
>>> ureg = pint.UnitRegistry()
>>> ureg.define('kilo- = 1000 = K- = k-')
>>> ureg.define('@alias meter = M')
>>> pvl_file = 'tests/data/pds3/units1.lbl'
>>> label = pvl.load(pvl_file, decoder=OmniDecoder(quantity_cls=ureg.Quantity))
>>> print(label)
PVLModule([
  ('PDS_VERSION_ID', 'PDS3')
  ('MSL:COMMENT', 'THING TEST')
  ('FLOAT_UNIT', <Quantity(0.414, 'kilometer')>)
  ('INT_UNIT', <Quantity(4, 'meter')>)
])
>>> print(type(label['FLOAT_UNIT']))
<class 'pint.quantity.build_quantity_class.<locals>.Quantity'>
```

Similarly, `pint.Quantity` objects can be encoded to PVL text by *`pvl.dump()`* or *`pvl.dumps()`*:

```
>>> import pvl
>>> import pint
>>> ureg = pint.UnitRegistry()
>>> dist = 15 * ureg.m
>>> vel = 0.5 * ureg.m / ureg.second
>>> my_label = dict(length=dist, velocity=vel)
>>> print(pvl.dumps(my_label))
LENGTH   = 15 <meter>
VELOCITY = 0.5 <meter / second>
END
```

CHAPTER 5

Utility Programs

This library also provides some command-line utility programs to work with PVL text.

5.1 pvl_translate

A program for converting PVL text to a specific PVL dialect.

The `pvl_translate` program will read a file with PVL text (any of the kinds of files that *`pvl.load()`* reads) or STDIN and will convert that PVL text to a particular PVL dialect. It is not particularly robust, and if it cannot make simple conversions, it will raise errors.

```
usage: pvl_translate [-h] -of {PDS3,ODL,ISIS,PVL,JSON} [--version]
                     [infile] [outfile]
```

`infile`
file containing PVL text to translate, defaults to STDIN.

`outfile`
file to write translated PVL to, defaults to STDOUT.

`-h, --help`
show this help message and exit

`-of` `{PDS3,ODL,ISIS,PVL,JSON}`, **`--output_format`** `{PDS3,ODL,ISIS,PVL,JSON}`
Select the format to create the new file as.

`--version`
show program's version number and exit

In the examples below will all operate on the file with these contents:

```
PDS_VERSION_ID       = PDS3

/* FILE DATA ELEMENTS */

```

(continues on next page)

(continued from previous page)

```
RECORD_TYPE           = FIXED_LENGTH
RECORD_BYTES          = 824
LABEL_RECORDS         = 1
FILE_RECORDS          = 601

/* POINTERS TO DATA OBJECTS */

^IMAGE                = 2

/* IMAGE DATA ELEMENTS */

OBJECT                = IMAGE
  LINES               = 600
  LINE_SAMPLES        = 824
  SAMPLE_TYPE         = MSB_INTEGER
  SAMPLE_BITS         = 8
  MEAN                = 51.67785396440129
  MEDIAN              = 50.00000
  MINIMUM             = 0
  MAXIMUM             = 255
  STANDARD_DEVIATION  = 16.97019
  CHECKSUM            = 25549531
END_OBJECT            = IMAGE

END
```

Convert to PDS3 (whitespace and comments get removed):

```
> pvl_translate -of PDS3 tests/data/pds3/simple_image_1.lbl
PDS_VERSION_ID = PDS3
RECORD_TYPE    = FIXED_LENGTH
RECORD_BYTES   = 824
LABEL_RECORDS  = 1
FILE_RECORDS   = 601
^IMAGE         = 2
OBJECT = IMAGE
  LINES               = 600
  LINE_SAMPLES        = 824
  SAMPLE_TYPE         = MSB_INTEGER
  SAMPLE_BITS         = 8
  MEAN                = 51.67785396440129
  MEDIAN              = 50.0
  MINIMUM             = 0
  MAXIMUM             = 255
  STANDARD_DEVIATION  = 16.97019
  CHECKSUM            = 25549531
END_OBJECT = IMAGE
END
```

Convert to PVL:

```
> pvl_translate -of PVL tests/data/pds3/simple_image_1.lbl
PDS_VERSION_ID = PDS3;
RECORD_TYPE    = FIXED_LENGTH;
RECORD_BYTES   = 824;
LABEL_RECORDS  = 1;
FILE_RECORDS   = 601;
```

(continues on next page)

(continued from previous page)

```
^IMAGE          = 2;
BEGIN_OBJECT = IMAGE;
  LINES               = 600;
  LINE_SAMPLES        = 824;
  SAMPLE_TYPE         = MSB_INTEGER;
  SAMPLE_BITS         = 8;
  MEAN                = 51.67785396440129;
  MEDIAN              = 50.0;
  MINIMUM             = 0;
  MAXIMUM             = 255;
  STANDARD_DEVIATION = 16.97019;
  CHECKSUM            = 25549531;
END_OBJECT = IMAGE;
END;
```

Convert to JSON:

```
> pvl_translate -of JSON tests/data/pds3/simple_image_1.lbl
{"PDS_VERSION_ID": "PDS3", "RECORD_TYPE": "FIXED_LENGTH", "RECORD_BYTES": 824, "LABEL_
↪RECORDS": 1, "FILE_RECORDS": 601, "^IMAGE": 2, "IMAGE": {"LINES": 600, "LINE_SAMPLES
↪": 824, "SAMPLE_TYPE": "MSB_INTEGER", "SAMPLE_BITS": 8, "MEAN": 51.67785396440129,
↪"MEDIAN": 50.0, "MINIMUM": 0, "MAXIMUM": 255, "STANDARD_DEVIATION": 16.97019,
↪"CHECKSUM": 25549531}}
```

5.2 pvl_validate

A program for testing and validating PVL text.

The `pvl_validate` program will read a file with PVL text (any of the kinds of files that *`pvl.load()`* reads) and will report on which of the various PVL dialects were able to load that PVL text, and then also reports on whether the `pvl` library can encode the Python Objects back out to PVL text.

You can imagine some PVL text that could be loaded, but is not able to be written out in a particular strict PVL dialect (like PDS3 labels).

```
usage: pvl_validate [-h] [-v] [--version] file [file ...]
```

`file`
: file containing PVL text to validate.

`-h, --help`
: show this help message and exit

`-v, --verbose`
: Will report the errors that are encountered. A second v will include tracebacks for non-pvl exceptions.

`--version`
: show program's version number and exit

Validate one file:

```
> pvl_validate tests/data/pds3/simple_image_1.lbl
PDS3  |     Loads     |     Encodes
ODL   |     Loads     |     Encodes
PVL   |     Loads     |     Encodes
```

(continues on next page)

(continued from previous page)

```
ISIS |      Loads      |      Encodes
Omni |      Loads      |      Encodes
>
```

You can see here that the `simple_image_1.lbl` file can be loaded and the resulting Python object encoded with each of the PVL dialects that the `pvl` library knows.

A file with broken PVL text:

```
> pvl_validate tests/data/pds3/broken/broken1.lbl
PDS3 | does NOT load |
ODL  | does NOT load |
PVL  | does NOT load |
ISIS |      Loads      |      Encodes
Omni |      Loads      |      Encodes
>
```

Here, the PVL text in broken1.lbl cannot be loaded by the PDS3, ODL, or PVL dialects, to learn why use `-v`:

```
> pvl_validate -v tests/data/pds3/broken/broken1.lbl
ERROR: PDS3 load error tests/data/pds3/broken/broken1.lbl (LexerError(...),
↪'Expecting an Aggregation Block, an Assignment Statement, or an End Statement, but␣
↪found "=" : line 3 column 7 (char 23)')
ERROR: ODL load error tests/data/pds3/broken/broken1.lbl (LexerError(...), 'Expecting␣
↪an Aggregation Block, an Assignment Statement, or an End Statement, but found "=" :␣
↪line 3 column 7 (char 23)')
ERROR: PVL load error tests/data/pds3/broken/broken1.lbl (LexerError(...), 'Expecting␣
↪an Aggregation Block, an Assignment Statement, or an End Statement, but found "=" :␣
↪line 3 column 7 (char 23)')
PDS3 | does NOT load |
ODL  | does NOT load |
PVL  | does NOT load |
ISIS |      Loads      |      Encodes
Omni |      Loads      |      Encodes
```

This tells us that in these cases, there is a parameter with a missing value. However, the OmniParser (the default, and also what the ISIS dialect uses) has more tolerance for broken PVL text, and is able to load it, and then write valid PVL back out.

Here's a file which has some PVL text which is valid for some dialects, but not others:

```
> pvl_validate tests/data/pds3/dates.lbl
PDS3 |      Loads      | does NOT encode
ODL  |      Loads      |      Encodes
PVL  |      Loads      |      Encodes
ISIS |      Loads      |      Encodes
Omni |      Loads      |      Encodes
>
```

Here, `pvl_validate` indicates that it can load the file with all of the PVL dialects, and can encode it back for most. What was the problem:

```
> pvl_validate -v tests/data/pds3/dates.lbl
ERROR: PDS3 encode error tests/data/pds3/dates.lbl PDS labels should only have UTC␣
↪times, but this time has a timezone: 01:12:22+07:00
PDS3 |      Loads      | does NOT encode
ODL  |      Loads      |      Encodes
```

(continues on next page)

(continued from previous page)

```
PVL   |      Loads      |      Encodes
ISIS  |      Loads      |      Encodes
Omni  |      Loads      |      Encodes
```

It indicates that it cannot encode the Python object out to the PDS3 format because it contains a date with a different time zone (which aren't allowed in a PDS3 Label). So this is an example of how the loaders are a little more permissive, but to really test whether some PVL text is conformant, it also should be able to be encoded.

In this case, if the user wants to write out a valid PDS3 label, they will have to do some work to fix the value.

Validating a bunch of files:

```
> pvl_validate tests/data/pds3/*lbl
---------------------------------------+----------+----------+----------+----------
↪-+----------
File                                   |   PDS3   |    ODL   |    PVL   |   ISIS  ␣
↪ |   Omni
---------------------------------------+----------+----------+----------+----------
↪-+----------
tests/data/pds3/backslashes.lbl        |  L    E  |  L    E  |  L    E  |  L    E ␣
↪ |  L    E
tests/data/pds3/based_integer1.lbl     |  L    E  |  L    E  |  L    E  |  L    E ␣
↪ |  L    E
tests/data/pds3/dates.lbl              |  L   No E |  L    E  |  L    E  |  L    E ␣
↪ |  L    E
tests/data/pds3/empty.lbl              |  L    E  |  L    E  |  L    E  |  L    E ␣
↪ |  L    E
tests/data/pds3/float1.lbl             |  L    E  |  L    E  |  L    E  |  L    E ␣
↪ |  L    E
tests/data/pds3/float_unit1.lbl        |  L    E  |  L    E  |  L    E  |  L    E ␣
↪ |  L    E
tests/data/pds3/group1.lbl             |  L    E  |  L    E  |  L    E  |  L    E ␣
↪ |  L    E
tests/data/pds3/group2.lbl             |  L    E  |  L    E  |  L    E  |  L    E ␣
↪ |  L    E
tests/data/pds3/group3.lbl             |  L    E  |  L    E  |  L    E  |  L    E ␣
↪ |  L    E
tests/data/pds3/group4.lbl             |  L    E  |  L    E  |  L    E  |  L    E ␣
↪ |  L    E
tests/data/pds3/namespaced_string1.lbl |  L    E  |  L    E  |  L    E  |  L    E ␣
↪ |  L    E
tests/data/pds3/negative_float1.lbl    |  L    E  |  L    E  |  L    E  |  L    E ␣
↪ |  L    E
tests/data/pds3/negative_int1.lbl      |  L    E  |  L    E  |  L    E  |  L    E ␣
↪ |  L    E
tests/data/pds3/nested_object1.lbl     |  L    E  |  L    E  |  L    E  |  L    E ␣
↪ |  L    E
tests/data/pds3/nested_object2.lbl     |  L    E  |  L    E  |  L    E  |  L    E ␣
↪ |  L    E
tests/data/pds3/scaled_real1.lbl       |  L    E  |  L    E  |  L    E  |  L    E ␣
↪ |  L    E
tests/data/pds3/sequence1.lbl          |  L    E  |  L    E  |  L    E  |  L    E ␣
↪ |  L    E
tests/data/pds3/sequence2.lbl          |  L    E  |  L    E  |  L    E  |  L    E ␣
↪ |  L    E
tests/data/pds3/sequence3.lbl          |  L    E  |  L    E  |  L    E  |  L    E ␣
↪ |  L    E
```

(continues on next page)

(continued from previous page)

```
tests/data/pds3/sequence_units1.lbl     |  L    E   |  L    E   |  L    E   |  L    E ␣
↪ |  L    E
tests/data/pds3/set1.lbl                |  L   No E |  L    E   |  L    E   |  L    E ␣
↪ |  L    E
tests/data/pds3/set2.lbl                |  L   No E |  L    E   |  L    E   |  L    E ␣
↪ |  L    E
tests/data/pds3/simple_image_1.lbl      |  L    E   |  L    E   |  L    E   |  L    E ␣
↪ |  L    E
tests/data/pds3/simple_image_2.lbl      |  L    E   |  L    E   |  L    E   |  L    E ␣
↪ |  L    E
tests/data/pds3/string2.lbl             |  L    E   |  L    E   |  L    E   |  L    E ␣
↪ |  L    E
tests/data/pds3/string3.lbl             |  L    E   |  L    E   |  L    E   |  L    E ␣
↪ |  L    E
tests/data/pds3/string4.lbl             |  L    E   |  L    E   |  L    E   |  L    E ␣
↪ |  L    E
tests/data/pds3/tiny1.lbl               |  L    E   |  L    E   |  L    E   |  L    E ␣
↪ |  L    E
tests/data/pds3/tiny2.lbl               |  L    E   |  L    E   |  L    E   |  L    E ␣
↪ |  L    E
tests/data/pds3/tiny3.lbl               |  L    E   |  L    E   |  L    E   |  L    E ␣
↪ |  L    E
tests/data/pds3/tiny4.lbl               |  L    E   |  L    E   |  L    E   |  L    E ␣
↪ |  L    E
tests/data/pds3/units1.lbl              |  L    E   |  L    E   |  L    E   |  L    E ␣
↪ |  L    E
tests/data/pds3/units2.lbl              |  L    E   |  L    E   |  L    E   |  L    E ␣
↪ |  L    E
>
```

and with -v:

```
> pvl_validate -v tests/data/pds3/*lbl
ERROR: PDS3 encode error tests/data/pds3/dates.lbl PDS labels should only have UTC␣
↪times, but this time has a timezone: 01:12:22+07:00
ERROR: PDS3 encode error tests/data/pds3/set1.lbl The PDS only allows integers and␣
↪symbols in sets: {1.5}
ERROR: PDS3 encode error tests/data/pds3/set2.lbl The PDS only allows integers and␣
↪symbols in sets: {2.33, 3.4}
-------------------------------------+----------+----------+----------+----------
↪-+----------
File                                 |   PDS3   |    ODL   |    PVL   |   ISIS   ␣
↪ |   Omni
-------------------------------------+----------+----------+----------+----------
↪-+----------
tests/data/pds3/backslashes.lbl      |  L    E  |  L    E  |  L    E  |  L    E ␣
↪ |  L    E
tests/data/pds3/based_integer1.lbl   |  L    E  |  L    E  |  L    E  |  L    E ␣
↪ |  L    E
tests/data/pds3/dates.lbl            |  L   No E|  L    E  |  L    E  |  L    E ␣
↪ |  L    E
tests/data/pds3/empty.lbl            |  L    E  |  L    E  |  L    E  |  L    E ␣
↪ |  L    E
[... output truncated ...]
```

CHAPTER 6

Standards & Specifications

Although many people use the term 'PVL' to describe parameter-value text, they are often unaware that there are at least four different 'dialects' or 'flavors' of 'PVL'. They are described below.

Unfortunately one of them is actually named PVL, so it is difficult to distinguish when someone is using "PVL" to refer to the formal specification versus when they are using "PVL" to refer to some text that could be parsable as PVL.

It is important to not that the `pvl` module's approach is: attempt to read everything, but write out standards-compliant PDS3 labels by default. That may not be the dialect of PVL-text that you want, and if so, you can easily change it by specifying the `encoder` parameter to the `pvl.dump()` or `pvl.dumps()` functions. In general, you could write `pvl.dump(somedictlike, encoder=pvl.encoder.PVLEncoder())` to dump out PVL-specification PVL-text. The options are: PVLEncoder(), ODLEncoder(), PDSLabelEncoder() (the default), and ISISEncoder().

In the documentation for this library, we will attempt to provide enough context for you to distinguish, but we will typically use "PVL text" to refer to some generic text that may or may not conform to one of the PVL 'dialects' or that could be converted into one of them. We will also use `pvl` to refer to this Python library.

In practice, since people are not typcially aware of the formal PVL specification, when most people say "PVL" they are most likely referring to generic "PVL text."

6.1 Parameter Value Language (PVL)

The definition of the Parameter Value Language (PVL) is based on the Consultive Committee for Space Data Systems, and their `Parameter Value Language Specification (CCSD0006 and CCSD0008), CCSDS 6441.0-B-2` referred to as the Blue Book with a date of June 2000.

This formal definition of PVL is quite permissive, and usually forms the base class of objects in this library.

6.2 Object Description Language (ODL)

The Object Description Language (ODL) is based on PVL, but adds additional restrictions. It is defined in the `PDS3 Standards Reference (version 3.8, 27 Feb 2009) Chapter 12:`

`Object Description Language Specification and Usage`.

However, even though ODL is specified by the PDS, by itself, it is not the definition that PDS3 labels should conform to. By and large, as a user, you are rarely interested in the ODL specification, and mostly want to deal with the PDS3 Standard.

6.3 PDS3 Standard

The PDS3 Standard is also defined in the PDS3 Standards Reference (version 3.8, 27 Feb 2009) Chapter 12: Object Description Language Specification. The PDS3 Standard are mostly additional restrictions on the base definition of ODL, and appear as additional notes or sections in the document.

6.4 ISIS Cube Label format

The ISIS software has used a custom implementation (through at least ISIS 3.9) to write PVL text into the labels of its cube files. This PVL text does not strictly follow any of the published standards. It was based on PDS3 ODL from the 1990s, but has some extensions adopted from existing and prior data sets from ISIS2, PDS, JAXA, ISRO, etc., and extensions used only within ISIS3 files (.cub, .net). This is one of the reasons using ISIS cube files as an archive format or PVL text written by ISIS as a submission to the PDS has been strongly discouraged.

Since there is no specification, only a detailed analysis of the ISIS software that writes its PVL text would yield a strategy for parsing it.

At this time, the loaders (*pvl.loads()* and *pvl.load()*) default to using the *pvl.parser.OmniParser* which should be able to parse most forms of PVL text that ISIS writes out or into its cube labels. However, this is very likely where a user could run into errors (if there is something that isn't supported), and we welcome bug reports to help extend our coverage of this flavor of PVL text.

CHAPTER 7

pvl

7.1 pvl package

7.1.1 Submodules

7.1.2 pvl.collections module

Parameter Value Language container datatypes providing enhancements to Python general purpose built-in containers.

To enable efficient operations on parsed PVL text, we need an object that acts as both a dict-like Mapping container and a list-like Sequence container, essentially an ordered multi-dict. There is no existing object or even an Abstract Base Class in the Python Standard Library for such an object. So we define the MutableMappingSequence ABC here, which is (as the name implies) an abstract base class that implements both the Python MutableMapping and Mutable Sequence ABCs. We also provide two implementations, the OrderedMultiDict, and the newer PVLMultiDict.

Additionally, for PVL Values which also have an associated PVL Units Expression, they need to be returned as a quantity object which contains both a notion of a value and the units for that value. Again, there is no fundamental Python type for a quantity, so we define the Quantity class (formerly the Units class).

class pvl.collections.**ItemsView**(*mapping*)
Bases: *pvl.collections.MappingView*

index(*item*)

class pvl.collections.**KeysView**(*mapping*)
Bases: *pvl.collections.MappingView*

index(*key*)

class pvl.collections.**MappingView**(*mapping*)
Bases: object

class pvl.collections.**MutableMappingSequence**
Bases: collections.abc.MutableMapping, collections.abc.MutableSequence

ABC for a mutable object that has both mapping and sequence characteristics.

Must implement *.getall(k)* and *.popall(k)* since a MutableMappingSequence can have many values for a single key, while *.get(k)* and *.pop(k)* return and operate on a single value, the *all* versions return and operate on all values in the MutableMappingSequence with the key *k*.

Furthermore, *.pop()* without an argument should function as the MutableSequence pop() function and pop the last value when considering the MutableMappingSequence in a list-like manner.

append(*key*, *value*)
S.append(value) – append value to the end of the sequence

getall(*key*)

popall(*key*)

class pvl.collections.**OrderedMultiDict**(**args*, ***kwargs*)
Bases: dict, *pvl.collections.MutableMappingSequence*

A dict like container.

This container preserves the original ordering as well as allows multiple values for the same key. It provides similar semantics to a list of tuples but with dict style access.

Using __setitem__ syntax overwrites all fields with the same key and __getitem__ will return the first value with the key.

append(*key*, *value*)
Adds a (name, value) pair, doesn't overwrite the value if it already exists.

clear() → None. Remove all items from D.

copy() → a shallow copy of D

discard(*key*)

extend(**args*, ***kwargs*)
Add key value pairs for an iterable.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

getall(*key*) → collections.abc.Sequence
Returns a list of all the values for a named field. Returns KeyError if the key doesn't exist.

getlist(*key*) → collections.abc.Sequence
Returns a list of all the values for the named field. Returns an empty list if the key doesn't exist.

insert(*index: int*, **args*) → None
Inserts at the index given by *index*.

The first positional argument will always be taken as the *index* for insertion.

If three arguments are given, the second will be taken as the *key*, and the third as the *value* to insert.

If only two arguments are given, the second must be a sequence.

If it is a sequence of pairs (such that every item in the sequence is itself a sequence of length two), that sequence will be inserted as key, value pairs.

If it happens to be a sequence of two items (the first of which is not a sequence), the first will be taken as the *key* and the second the *value* to insert.

insert_after(*key*, *new_item: collections.abc.Iterable*, *instance=0*)
Insert an item after a key

insert_before(*key*, *new_item: collections.abc.Iterable*, *instance=0*)
Insert an item before a key

items() → a set-like object providing a view on D's items

key_index(*key*, *instance: int = 0*) → int
Get the index of the key to insert before or after.

keys() → a set-like object providing a view on D's keys

pop(**args*, ***kwargs*)
Removes all items with the specified *key*.

popall(*key*, *default=<object object>*)
D.pop(k[,d]) -> v, remove specified key and return the corresponding value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

update(*[E]*, ***F*) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values() → an object providing a view on D's values

class pvl.collections.**PVLAggregation**(**args*, ***kwargs*)
Bases: *pvl.collections.OrderedMultiDict*

class pvl.collections.**PVLGroup**(**args*, ***kwargs*)
Bases: *pvl.collections.PVLAggregation*

class pvl.collections.**PVLModule**(**args*, ***kwargs*)
Bases: *pvl.collections.OrderedMultiDict*

class pvl.collections.**PVLObject**(**args*, ***kwargs*)
Bases: *pvl.collections.PVLAggregation*

class pvl.collections.**Quantity**
Bases: *pvl.collections.Quantity*

A simple collections.namedtuple object to contain a value and units parameter.

If you need more comprehensive units handling, you may want to use the astropy.units.Quantity object, the pint.Quantity object, or some other 3rd party object. Please see the documentation on *Quantities: Values and Units* for how to use 3rd party Quantity objects with pvl.

class pvl.collections.**Units**
Bases: *pvl.collections.Quantity*

class pvl.collections.**ValuesView**(*mapping*)
Bases: *pvl.collections.MappingView*

index(*value*)

pvl.collections.**dict_delitem**
Delete self[key].

pvl.collections.**dict_setitem**
Set self[key] to value.

7.1.3 pvl.decoder module

Parameter Value Language decoder.

The definition of PVL used in this module is based on the Consultive Committee for Space Data Systems, and their Parameter Value Language Specification (CCSD0006 and CCSD0008), CCSDS 6441.0-B-2, referred to as the Blue Book with a date of June 2000.

A decoder deals with converting strings given to it (typically by the parser) to the appropriate Python type.

class pvl.decoder.**ODLDecoder**(*grammar=None*, *quantity_cls=None*, *real_cls=None*)
Bases: *pvl.decoder.PVLDecoder*

A decoder based on the rules in the PDS3 Standards Reference (version 3.8, 27 Feb 2009) Chapter 12: Object Description Language Specification and Usage.

Extends PVLDecoder, and if *grammar* is not specified, it will default to an ODLGrammar() object.

decode_datetime(*value: str*)
Extends parent function to also deal with datetimes and times with a time zone offset.

If it cannot, it will raise a ValueError.

decode_non_decimal(*value: str*) → int
Extends parent function by allowing the wider variety of radix values that ODL permits over PVL.

decode_quoted_string(*value: str*) → str
Extends parent function because the ODL specification allows for a dash (-) line continuation character that results in the dash, the line end, and any leading whitespace on the next line to be removed. It also allows for a sequence of format effectors surrounded by spacing characters to be collapsed to a single space.

decode_unquoted_string(*value: str*) → str
Extends parent function to provide the extra enforcement that only ODL Identifier text may be unquoted as a value.

static is_identifier(*value*)
Returns true if *value* is an ODL Identifier, false otherwise.

An ODL Identifier is composed of letters, digits, and underscores. The first character must be a letter, and the last must not be an underscore.

class pvl.decoder.**OmniDecoder**(*grammar=None*, *quantity_cls=None*, *real_cls=None*)
Bases: *pvl.decoder.ODLDecoder*

A permissive decoder that attempts to parse all forms of "PVL" that are thrown at it.

Extends ODLDecoder.

decode_datetime(*value: str*)
Returns an appropriate Python datetime time, date, or datetime object by using the 3rd party dateutil library (if present) to parse an ISO 8601 datetime string in *value*. If it cannot, or the dateutil library is not present, it will raise a ValueError.

decode_non_decimal(*value: str*) → int
Extends parent function by allowing a plus or minus sign to be in two different positions in a non-decimal number, since PVL has one specification, and ODL has another.

decode_unquoted_string(*value: str*) → str
Overrides parent function since the ODLDecoder has a more narrow definition of what is allowable as an unquoted string than the PVLDecoder does.

class pvl.decoder.**PDSLabelDecoder**(*grammar=None*, *quantity_cls=None*)
Bases: *pvl.decoder.ODLDecoder*

A decoder based on the rules in the PDS3 Standards Reference (version 3.8, 27 Feb 2009) Chapter 12: Object Description Language Specification and Usage.

Extends ODLDecoder, and if *grammar* is not specified, it will default to a PDS3Grammar() object.

decode_datetime(*value: str*)
Overrides parent function since PDS3 forbids a timezone specification, and times with a precision more than miliseconds.

If it cannot decode properly, it will raise a ValueError.

class pvl.decoder.**PVLDecoder**(*grammar=None*, *quantity_cls=None*, *real_cls=None*)
Bases: object

A decoder based on the rules in the CCSDS-641.0-B-2 'Blue Book' which defines the PVL language.

Parameters

- **grammar** – defaults to a *pvl.grammar.PVLGrammar*, but can be any object that implements the *pvl.grammar* interface.
- **quantity_cls** – defaults to *pvl.collections.Quantity*, but could be any class object that takes two arguments, where the first is the value, and the second is the units value.
- **real_cls** – defaults to float, but could be any class object that can be constructed from a *str* object.

decode(*value: str*)
Returns a Python object based on *value*.

decode_datetime(*value: str*)
Takes a string and attempts to convert it to the appropriate Python datetime time, date, or datetime type based on this decoder's grammar, or in one case, a str.

The PVL standard allows for the seconds value to range from zero to 60, so that the 60 can accommodate leap seconds. However, the Python datetime classes don't support second values for more than 59 seconds.

If a time with 60 seconds is encountered, it will not be returned as a datetime object (since that is not representable via Python datetime objects), but simply as a string.

The user can then then try and use the time module to parse this string into a time.struct_time. We chose not to do this with pvl because time.struct_time is a full *datetime* like object, even if it parsed only a *time* like object, the year, month, and day values in the time.struct_time would default, which could be misleading.

Alternately, the pvl.grammar.PVLGrammar class contains two regexes: leap_second_Ymd_re and leap_second_Yj_re which could be used along with the re.match object's groupdict() function to extract the string representations of the various numerical values, cast them to the appropriate numerical types, and do something useful with them.

decode_decimal(*value: str*)
Returns a Python int or self.real_cls object, as appropriate based on *value*. Raises a ValueError otherwise.

decode_non_decimal(*value: str*) → int
Returns a Python int as decoded from *value* on the assumption that *value* conforms to a non-decimal integer value as defined by this decoder's grammar, raises ValueError otherwise.

decode_quantity(*value*, *unit*)
Returns a Python object that represents a value with an associated unit, based on the values provided via *value* and *unit*. This function creates an object based on the decoder's *quantity_cls*.

decode_quoted_string(*value: str*) → str
Returns a Python `str` if *value* begins and ends with matching quote characters based on this decoder's grammar. Raises ValueError otherwise.

decode_simple_value(*value: str*)
Returns a Python object based on *value*, assuming that *value* can be decoded as a PVL Simple Value:

```
<Simple-Value> ::= (<Date-Time> | <Numeric> | <String>)
```

decode_unquoted_string(*value: str*) → str
Returns a Python `str` if *value* can be decoded as an unquoted string, based on this decoder's grammar. Raises a ValueError otherwise.

is_leap_seconds(*value: str*) → bool
Returns True if *value* is a time that matches the grammar's definition of a leap seconds time (a time string with a value of 60 for the seconds value). False otherwise.

pvl.decoder.**for_try_except**(*exception*, *function*, **iterable*)
Return the result of the first successful application of *function* to an element of *iterable*. If the *function* raises an Exception of type *exception*, it will continue to the next item of *iterable*. If there are no successful applications an Exception of type *exception* will be raised.

If additional *iterable* arguments are passed, *function* must take that many arguments and is applied to the items from all iterables in parallel (like `map()`). With multiple iterables, the iterator stops when the shortest iterable is exhausted.

7.1.4 pvl.encoder module

Parameter Value Langage encoder.

An encoder deals with converting Python objects into string values that conform to a PVL specification.

class pvl.encoder.**ISISEncoder**(*grammar=None*, *decoder=None*, *indent=2*, *width=80*, *aggregation_end=True*, *end_delimiter=False*, *newline='n'*, *group_class=<class 'pvl.collections.PVLGroup'>*, *object_class=<class 'pvl.collections.PVLObject'>*)
Bases: *pvl.encoder.PVLEncoder*

An encoder for writing PVL text that can be parsed by the ISIS PVL text parser.

The ISIS3 implementation (as of 3.9) of PVL/ODL (like) does not strictly follow any of the published standards. It was based on PDS3 ODL from the 1990s, but has several extensions adopted from existing and prior data sets from ISIS2, PDS, JAXA, ISRO, ..., and extensions used only within ISIS files (cub, net). This is one of the reasons using ISIS cube files or PVL text written by ISIS as an archive format has been strongly discouraged.

Since there is no specification, only a detailed analysis of the ISIS software that parses and writes its PVL text would yield a strategy for parsing it. This encoder is most likely the least reliable for that reason. We welcome bug reports to help extend our coverage of this flavor of PVL text.

Parameters

- **grammar** – defaults to pvl.grammar.ISISGrammar().
- **decoder** – defaults to pvl.decoder.PVLDecoder().
- **end_delimiter** – defaults to False.
- **newline** – defaults to '\n'.

class pvl.encoder.**ODLEncoder**(*grammar=None*, *decoder=None*, *indent=2*, *width=80*, *aggregation_end=True*, *end_delimiter=False*, *newline='rn'*, *group_class=<class 'pvl.collections.PVLGroup'>*, *object_class=<class 'pvl.collections.PVLObject'>*)

Bases: *pvl.encoder.PVLEncoder*

An encoder based on the rules in the PDS3 Standards Reference (version 3.8, 27 Feb 2009) Chapter 12: Object Description Language Specification and Usage for ODL only. This is almost certainly not what you want. There are very rarely cases where you'd want to use ODL that you wouldn't also want to use the PDS Label restrictions, so you probably really want the PDSLabelEncoder class, not this one. Move along.

It extends PVLEncoder.

Parameters

- **grammar** – defaults to pvl.grammar.ODLGrammar().
- **decoder** – defaults to pvl.decoder.ODLDecoder().
- **end_delimiter** – defaults to False.
- **newline** – defaults to '\r\n'.

encode(*module: collections.abc.Mapping*) → str
Extends parent function, but ODL requires that there must be a spacing or format character after the END statement and this adds the encoder's `newline` sequence.

encode_assignment(*key*, *value*, *level=0*, *key_len=None*) → str
Overrides parent function by restricting the length of keywords and enforcing that they be ODL Identifiers and uppercasing their characters.

encode_sequence(*value*) → str
Extends parent function, as ODL only allows one- and two-dimensional sequences of ODL scalar_values.

encode_set(*values*) → str
Extends parent function, ODL only allows sets to contain scalar values.

encode_string(*value*)
Extends parent function by appropriately quoting Symbol Strings.

encode_time(*value: datetime.time*) → str
Extends parent function since ODL allows a time zone offset from UTC to be included, and otherwise recommends that times be suffixed with a 'Z' to clearly indicate that they are in UTC.

encode_units(*value*) → str
Overrides parent function since ODL limits what characters and operators can be present in Units Expressions.

encode_value(*value*)
Extends parent function by only allowing Units Expressions for numeric values.

is_assignment_statement(*s*) → bool
Returns true if *s* is an ODL Assignment Statement, false otherwise.

An ODL Assignment Statement is either an element_identifier or a namespace_identifier joined to an element_identifier with a colon.

is_scalar(*value*) → bool
Returns a boolean indicating whether the *value* object qualifies as an ODL 'scalar_value'.

ODL defines a 'scalar-value' as a numeric_value, a date_time_string, a text_string_value, or a symbol_value.

For Python, these correspond to the following:

- numeric_value: any of self.numeric_types, and Quantity whose value is one of the self.numeric_types.
- date_time_string: datetime objects
- text_string_value: str
- symbol_value: str

is_symbol(*value*) → bool
Returns true if *value* is an ODL Symbol String, false otherwise.

An ODL Symbol String is enclosed by single quotes and may not contain any of the following characters:

1. The apostrophe, which is reserved as the symbol string delimiter.
2. ODL Format Effectors
3. Control characters

This means that an ODL Symbol String is a subset of the PVL quoted string, and will be represented in Python as a `str`.

needs_quotes(*s: str*) → bool
Return true if *s* is an ODL Identifier, false otherwise.

Overrides parent function.

class pvl.encoder.**PDSLabelEncoder**(*grammar=None*, *decoder=None*, *indent=2*, *width=80*, *aggregation_end=True*, *group_class=<class 'pvl.collections.PVLGroup'>*, *object_class=<class 'pvl.collections.PVLObject'>*, *convert_group_to_object=True*, *tab_replace=4*, *symbol_single_quote=True*, *time_trailing_z=True*)
Bases: *pvl.encoder.ODLEncoder*

An encoder based on the rules in the PDS3 Standards Reference (version 3.8, 27 Feb 2009) Chapter 12: Object Description Language Specification and Usage and writes out labels that conform to the PDS 3 standards.

It extends ODLEncoder.

You are not allowed to chose *end_delimiter* or *newline* as the parent class allows, because to be PDS-compliant, those are fixed choices. However, in some cases, the PDS3 Standards are asymmetric, allowing for a wider variety of PVL-text on "read" and a more narrow variety of PVL-text on "write". The default values of the PDSLabelEncoder enforce those strict "write" rules, but if you wish to alter them, but still produce PVL-text that would validate against the PDS3 standard, you may alter them.

Parameters

- **convert_group_to_object** – Defaults to True, meaning that if a GROUP does not conform to the PDS definition of a GROUP, then it will be written out as an OBJECT. If it is False, then an exception will be thrown if incompatible GROUPs are encountered. In PVL and ODL, the OBJECT and GROUP aggregations are interchangeable, but the PDS applies restrictions to what can appear in a GROUP.
- **tab_replace** – Defaults to 4 and indicates the number of space characters to replace horizontal tab characters with (since tabs aren't allowed in PDS labels). If this is set to zero, tabs will not be replaced with spaces.
- **symbol_single_quotes** – Defaults to True, and if a Python *str* object qualifies as a PVL Symbol String, it will be written to PVL-text as a single-quoted string. If False, no special handling is done, and any PVL Symbol String will be treated as a PVL Text String, which is typically enclosed with double-quotes.

- **time_trailing_z** – defaults to True, and suffixes a "Z" to datetimes and times written to PVL-text as the PDS encoding standard requires. If False, no trailing "Z" is written.

count_aggs(*module: collections.abc.Mapping*, *obj_count: int = 0*, *grp_count: int = 0) -> (<class 'int'>, <class 'int'>*)
Returns the count of OBJECT and GROUP aggregations that are contained within the *module* as a two-tuple in that order.

encode(*module: collections.abc.MutableMapping*) → str
Extends the parent function, by adding a restriction. For PDS, if there are any GROUP elements, there must be at least one OBJECT element in the label. Behavior here depends on the value of this encoder's convert_group_to_object property.

encode_aggregation_block(*key*, *value*, *level=0*)
Extends parent function because PDS has restrictions on what may be in a GROUP.

If the encoder's *convert_group_to_object* parameter is True, and a GROUP does not conform to the PDS definition of a GROUP, then it will be written out as an OBJECT. If it is False, then an exception will be thrown.

encode_set(*values*) → str
Extends parent function because PDS only allows symbol values and integers within sets.

encode_string(*value*)
Extends parent function to treat Symbol Strings as Text Strings, which typically means that they are double-quoted and not single-quoted.

encode_time(*value: datetime.time*) → str
Overrides parent's encode_time() function because even though ODL allows for timezones, PDS does not.

Not in the section on times, but at the end of the PDS ODL document, in section 12.7.3, para 14, it indicates that alternate time zones may not be used in a PDS label, only these: 1. YYYY-MM-DDTHH:MM:SS.SSS. 2. YYYY-DDDTHH:MM:SS.SSS.

is_PDSgroup(*group: collections.abc.Mapping*) → bool
Returns true if the dict-like *group* qualifies as a PDS Group, false otherwise.

PDS applies the following restrictions to GROUPS:

1. The GROUP structure may only be used in a data product label which also contains one or more data OBJECT definitions.
2. The GROUP statement must contain only attribute assignment statements, include pointers, or related information pointers (i.e., no data location pointers). If there are multiple values, a single statement must be used with either sequence or set syntax; no attribute assignment statement or pointer may be repeated.
3. GROUP statements may not be nested.
4. GROUP statements may not contain OBJECT definitions.
5. Only PSDD elements may appear within a GROUP statement. *PSDD is not defined anywhere in the PDS document, so don't know how to test for it.*
6. The keyword contents associated with a specific GROUP identifier must be identical across all labels of a single data set (with the exception of the "PARAMETERS" GROUP, as explained).

Use of the GROUP structure must be coordinated with the responsible PDS discipline Node.

Items 1 & 6 and the final sentence above, can't really be tested by examining a single group, but must be dealt with in a larger context. The ODLEncoder.encode_module() handles #1, at least. You're on your own for the other two issues.

Item 5: *PSDD* is not defined anywhere in the ODL PDS document, so don't know how to test for it.

class pvl.encoder.**PVLEncoder**(*grammar=None*, *decoder=None*, *indent: int = 2*, *width: int = 80*, *aggregation_end: bool = True*, *end_delimiter: bool = True*, *newline: str = 'n'*, *group_class=<class 'pvl.collections.PVLGroup'>*, *object_class=<class 'pvl.collections.PVLObject'>*)

Bases: object

An encoder based on the rules in the CCSDS-641.0-B-2 'Blue Book' which defines the PVL language.

Parameters

- **grammar** – A pvl.grammar object, if None or not specified, it will be set to the grammar parameter of *decoder* (if *decoder* is not None) or will default to PVLGrammar().
- **grammar** – defaults to pvl.grammar.PVLGrammar().
- **decoder** – defaults to pvl.decoder.PVLDecoder().
- **indent** – specifies the number of spaces that will be used to indent each level of the PVL document, when Groups or Objects are encountered, defaults to 2.
- **width** – specifies the number of characters in width that each line should have, defaults to 80.
- **aggregation_end** – when True the encoder will print the value of the aggregation's Block Name in the End Aggregation Statement (e.g. END_GROUP = foo), and when false, it won't (e.g. END_GROUP). Defaults to True.
- **end_delimiter** – when True the encoder will print the grammar's delimiter (e.g. ';' for PVL) after each statement, when False it won't. Defaults to True.
- **newline** – is the string that will be placed at the end of each 'line' of output (and counts against *width*), defaults to '\n'.
- **group_class** – must this class will be tested against with isinstance() to determine if various elements of the dict-like passed to encode() should be encoded as a PVL Group or PVL Object, defaults to PVLGroup.
- **object_class** – must be a class that can take a *group_class* object in its constructor (essentially converting a *group_class* to an *object_class*), otherwise will raise TypeError. Defaults to PVLObject.

add_quantity_cls(*cls*, *value_prop: str*, *units_prop: str*)

Adds a quantity class to the list of possible quantities that this encoder can handle.

Parameters

- **cls** – The name of a quantity class that can be tested with `isinstance()`.
- **value_prop** – A string that is the property name of *cls* that contains the value or magnitude of the quantity object.
- **units_prop** – A string that is the property name of *cls* that contains the units element of the quantity object.

encode(*module: collections.abc.Mapping*) → str

Returns a `str` formatted as a PVL document based on the dict-like *module* object according to the rules of this encoder.

encode_aggregation_block(*key: str*, *value: collections.abc.Mapping*, *level: int = 0*) → str

Returns a `str` formatted as a PVL Aggregation Block with *key* as its name, and its contents based on the dict-like *value* object according to the rules of this encoder, with an indentation level of *level*.

encode_assignment(*key: str*, *value*, *level: int = 0*, *key_len: int = None*) → str

Returns a `str` formatted as a PVL Assignment Statement with *key* as its Parameter Name, and its value

based on *value* object according to the rules of this encoder, with an indentation level of *level*. It also allows for an optional *key_len* which indicates the width in characters that the Assignment Statement should be set to, defaults to the width of *key*.

static encode_date(*value: datetime.date*) → str
Returns a `str` formatted as a PVL Date based on the *value* object according to the rules of this encoder.

encode_datetime(*value: datetime.datetime*) → str
Returns a `str` formatted as a PVL Date/Time based on the *value* object according to the rules of this encoder.

encode_datetype(*value*) → str
Returns a `str` formatted as a PVL Date/Time based on the *value* object according to the rules of this encoder. If *value* is not a datetime date, time, or datetime object, it will raise TypeError.

encode_module(*module: collections.abc.Mapping*, *level: int = 0*) → str
Returns a `str` formatted as a PVL module based on the dict-like *module* object according to the rules of this encoder, with an indentation level of *level*.

encode_quantity(*value*) → str
Returns a `str` formatted as a PVL Value followed by a PVL Units Expression if the *value* object can be encoded this way, otherwise raise ValueError.

encode_sequence(*value: collections.abc.Sequence*) → str
Returns a `str` formatted as a PVL Sequence based on the *value* object according to the rules of this encoder.

encode_set(*value: collections.abc.Set*) → str
Returns a `str` formatted as a PVL Set based on the *value* object according to the rules of this encoder.

encode_setseq(*values: collections.abc.Collection*) → str
This function provides shared functionality for encode_sequence() and encode_set().

encode_simple_value(*value*) → str
Returns a `str` formatted as a PVL Simple Value based on the *value* object according to the rules of this encoder.

encode_string(*value*) → str
Returns a `str` formatted as a PVL String based on the *value* object according to the rules of this encoder.

static encode_time(*value: datetime.time*) → str
Returns a `str` formatted as a PVL Time based on the *value* object according to the rules of this encoder.

encode_units(*value: str*) → str
Returns a `str` formatted as a PVL Units Value based on the *value* object according to the rules of this encoder.

encode_value(*value*) → str
Returns a `str` formatted as a PVL Value based on the *value* object according to the rules of this encoder.

encode_value_units(*value*, *units*) → str
Returns a `str` formatted as a PVL Value from *value* followed by a PVL Units Expressions from *units*.

format(*s: str*, *level: int = 0*) → str
Returns a string derived from *s*, which has leading space characters equal to *level* times the number of spaces specified by this encoder's indent property.

It uses the textwrap library to wrap long lines.

needs_quotes(*s: str*) → bool
Returns true if *s* must be quoted according to this encoder's grammar, false otherwise.

class pvl.encoder.**QuantTup**
Bases: *pvl.encoder.QuantTup*

This class is just a convenient namedtuple for internally keeping track of quantity classes that encoders can deal with. In general, users should not be instantiating this, instead use your encoder's add_quantity_cls() function.

7.1.5 pvl.exceptions module

Exceptions for the Parameter Value Library.

exception pvl.exceptions.**LexerError**(*msg*, *doc*, *pos*, *lexeme*)
Bases: ValueError

Subclass of ValueError with the following additional properties:

msg: The unformatted error message doc: The PVL text being parsed pos: The start index in doc where parsing failed lineno: The line corresponding to pos colno: The column corresponding to pos

exception pvl.exceptions.**ParseError**(*msg*, *token=None*)
Bases: Exception

An exception to signal errors in the pvl parser.

exception pvl.exceptions.**QuantityError**
Bases: Exception

A simple exception to distinguish errors from Quantity classes.

pvl.exceptions.**firstpos**(*sub: str*, *pos: int*)
On the assumption that *sub* is a substring contained in a longer string, and *pos* is the index in that longer string of the final character in sub, returns the position of the first character of sub in that longer string.

This is useful in the PVL library when we know the position of the final character of a token, but want the position of the first character.

pvl.exceptions.**linecount**(*doc: str*, *end: int*, *start: int = 0*)
Returns the number of lines (by counting the number of newline characters n, with the first line being line number one) in the string *doc* between the positions *start* and *end*.

7.1.6 pvl.grammar module

Describes the language aspects of PVL dialects.

These grammar objects are not particularly meant to be easily user-modifiable during running of an external program, which is why they have no arguments at initiation time, nor are there any methods or functions to modify them. This is because these grammar objects are used both for reading and writing PVL-text. As such, objects like PVLGrammar and ODLGrammar shouldn't be altered, because if they are, then the PVL-text written out with them wouldn't conform to the spec.

Certainly, these objects do have attributes that can be altered, but unless you've carefully read the code, it isn't recommended.

Maybe someday we'll add a more user-friendly interface to allow that, but in the meantime, just leave an Issue on the GitHub repo.

class pvl.grammar.**ISISGrammar**
Bases: *pvl.grammar.PVLGrammar*

This defines the ISIS version of PVL.

This is valid as of ISIS 3.9, and before, at least.

The ISIS 'Pvl' object typically writes out parameter values and keywords in CamelCase (e.g. 'Group', 'End_Group', 'CenterLatitude', etc.), but it will accept all-uppercase versions.

Technically, since the ISIS 'Pvl' object which parses PVL text into C++ objects for ISIS programs to work with does not recognize the 'BEGIN_<GROUP|OBJECT>' construction, this means that ISIS does not parse PVL text that would be valid according to the PVL, ODL, or PDS3 specs.

static adjust_reserved_characters(*chars: collections.abc.Iterable*)

comments = (('/*', '*/'), ('#', '\n'))

group_keywords = {'GROUP': 'END_GROUP'}

group_pref_keywords = ('Group', 'End_Group')

object_keywords = {'OBJECT': 'END_OBJECT'}

object_pref_keywords = ('Object', 'End_Object')

class pvl.grammar.**ODLGrammar**

Bases: *pvl.grammar.PVLGrammar*

This defines an ODL grammar.

The reference for this grammar is the PDS3 Standards Reference (version 3.8, 27 Feb 2009) Chapter 12: Object Description Language Specification and Usage.

char_allowed(*char*)

Returns true if *char* is allowed in the ODL Character Set.

The ODL Character Set is limited to ASCII. This is fewer characters than PVL, but appears to allow more control characters to be in quoted strings than PVL does.

default_timezone = None

group_pref_keywords = ('GROUP', 'END_GROUP')

leap_second_Yj_re = None

leap_second_Ymd_re = None

nondecimal_pre_re = re.compile('(?P<radix>[2-9]|1[0-6])#(?P<sign>[+-]?)')

nondecimal_re = re.compile('(?P<radix>[2-9]|1[0-6])#(?P<sign>[+-]?)(?P<non_decimal>[0-

object_pref_keywords = ('OBJECT', 'END_OBJECT')

class pvl.grammar.**OmniGrammar**

Bases: *pvl.grammar.PVLGrammar*

A broadly permissive grammar.

This grammar does not follow a specification, but is meant to allow the broadest possible ingestion of PVL-like text that is found.

This grammar should not be used to write out Python objects to PVL, instead please use one of the grammars that follows a published specification, like the PVLGrammar or the ODLGrammar.

char_allowed(*char*)

Takes all characters, could accept bad things, and the user must beware.

comments = (('/*', '*/'), ('#', '\n'))

nondecimal_pre_re = re.compile('(?P<sign>[+-]?)(?P<radix>[2-9]|1[0-6])#(?P<second_sign

nondecimal_re = re.compile('(?P<sign>[+-]?)(?P<radix>[2-9]|1[0-6])#(?P<second_sign>[+-

class pvl.grammar.**PDSGrammar**
Bases: *pvl.grammar.ODLGrammar*

This defines a PDS3 ODL grammar.

The reference for this grammar is the PDS3 Standards Reference (version 3.8, 27 Feb 2009) Chapter 12: Object Description Language Specification and Usage.

default_timezone = datetime.timezone.utc

class pvl.grammar.**PVLGrammar**
Bases: object

Describes a PVL grammar for use by the lexer and parser.

The reference for this grammar is the CCSDS-641.0-B-2 'Blue Book'.

aggregation_keywords = {'BEGIN_GROUP': 'END_GROUP', 'BEGIN_OBJECT': 'END_OBJECT', 'GROU

binary_re = re.compile('(?P<sign>[+-]?)(?P<radix>2)#(?P<non_decimal>[01]+)#')

char_allowed(*char*)
Returns true if *char* is allowed in the PVL Character Set.

This is defined as most of the ISO 8859-1 'latin-1' character set with some exclusions.

comments = (('/*', '*/'),)

d = '%Y-%j'

date_formats = ('%Y-%m-%d', '%Y-%j', '%Y-%m-%dZ', '%Y-%jZ')

datetime_formats = ['%Y-%m-%dT%H:%M', '%Y-%m-%dT%H:%MZ', '%Y-%m-%dT%H:%M:%S', '%Y-%m-%

default_timezone = datetime.timezone.utc

delimiters = (';',)

end_statements = ('END',)

false_keyword = 'FALSE'

format_effectors = ('\n', '\r', '\x0b', '\x0c')

group_keywords = {'BEGIN_GROUP': 'END_GROUP', 'GROUP': 'END_GROUP'}

group_pref_keywords = ('BEGIN_GROUP', 'END_GROUP')

hex_re = re.compile('(?P<sign>[+-]?)(?P<radix>16)#(?P<non_decimal>[0-9A-Fa-f]+)#')

leap_second_Yj_re = re.compile('((?P<year>\\d{3}[1-9])-(?P<doy>(00[1-9]|0[1-9]\\d)|[12

leap_second_Ymd_re = re.compile('((?P<year>\\d{3}[1-9])-(?P<month>0[1-9]|1[0-2])-(?P<d

nondecimal_pre_re = re.compile('(?P<sign>[+-]?)(?P<radix>2|8|16)#')

nondecimal_re = re.compile('(?P<sign>[+-]?)(?P<radix>2|8|16)#(?P<non_decimal>[0-9|A-Fa

none_keyword = 'NULL'

numeric_start_chars = ('+', '-')

object_keywords = {'BEGIN_OBJECT': 'END_OBJECT', 'OBJECT': 'END_OBJECT'}

object_pref_keywords = ('BEGIN_OBJECT', 'END_OBJECT')

octal_re = re.compile('(?P<sign>[+-]?)(?P<radix>8)#(?P<non_decimal>[0-7]+)#')

p = ('BEGIN_OBJECT', 'END_OBJECT')

```
quotes = ('"', "'")
reserved_characters = ('&', '<', '>', "'", '{', '}', ',', '[', ']', '=', '!', '#', '('
reserved_keywords = {'BEGIN_GROUP', 'BEGIN_OBJECT', 'END', 'END_GROUP', 'END_OBJECT',
sequence_delimiters = ('(', ')')
set_delimiters = ('{', '}')
spacing_characters = (' ', '\t')
t = '%H:%M:%S.%f'
time_formats = ('%H:%M', '%H:%M:%S', '%H:%M:%S.%f', '%H:%MZ', '%H:%M:%SZ', '%H:%M:%S.%
true_keyword = 'TRUE'
units_delimiters = ('<', '>')
whitespace = (' ', '\t', '\n', '\r', '\x0b', '\x0c')
```

7.1.7 pvl.lexer module

Provides lexer functions for PVL.

class pvl.lexer.**Preserve**

Bases: enum.Enum

An enumeration.

COMMENT = 2

FALSE = 1

NONDECIMAL = 5

QUOTE = 4

UNIT = 3

pvl.lexer.**lex_char**(*char: str*, *prev_char: str*, *next_char: str*, *lexeme: str*, *preserve: dict*, *g: pvl.grammar.PVLGrammar*, *c_info: dict) -> (<class 'str'>, <class 'dict'>*)

Returns a modified *lexeme* string and a modified *preserve* dict in a two-tuple.

This is the main lexer() helper function for determining how to modify (or not) *lexeme* and *preserve* based on the single character in *char* and the other values passed into this function.

pvl.lexer.**lex_comment**(*char: str*, *prev_char: str*, *next_char: str*, *lexeme: str*, *preserve: dict*, *c_info: dict) -> (<class 'str'>, <class 'dict'>*)

Returns a modified *lexeme* string and a modified *preserve* dict in a two-tuple.

This is a lexer() helper function for determining how to modify *lexeme* and *preserve* based on the single character in *char* which may or may not be a comment character.

This function just makes the decision about whether to call lex_multichar_comments() or lex_singlechar_comments(), and then returns what they return.

pvl.lexer.**lex_continue**(*char: str*, *next_char: str*, *lexeme: str*, *token: pvl.token.Token*, *preserve: dict*, *g: pvl.grammar.PVLGrammar*) → bool

Return True if accumulation of *lexeme* should continue based on the values passed into this function, false otherwise.

This is a lexer() helper function.

pvl.lexer.**lex_multichar_comments**(*char: str*, *prev_char: str*, *next_char: str*, *lexeme: str*, *preserve: dict*, *comments: (<class 'str'>, <class 'str'>) = (('/*', '*/'),)) -> (<class 'str'>, <class 'dict'>)*

Returns a modified *lexeme* string and a modified *preserve* dict in a two-tuple.

This is a lexer() helper function for determining how to modify *lexeme* and *preserve* based on the single character in *char* which may or may not be part of a multi-character comment character group.

This function has an internal list of allowed pairs of multi-character comments that it can deal with, if the *comments* tuple contains any two-tuples that cannot be handled, a NotImplementedError will be raised.

This function will determine whether to append *char* to *lexeme* or not, and will set the value of the 'state' and 'end' values of *preserve* appropriately.

pvl.lexer.**lex_preserve**(*char: str*, *lexeme: str*, *preserve: dict) -> (<class 'str'>, <class 'dict'>)*

Returns a modified *lexeme* string and a modified *preserve* dict in a two-tuple. The modified *lexeme* will always be the concatenation of *lexeme* and *char*.

This is a lexer() helper function that is responsible for changing the state of the *preserve* dict, if needed.

If the value for 'end' in *preserve* is the same as *char*, then the modified *preserve* will have its 'state' value set to `Preserve.FALSE` and its 'end' value set to None, otherwise second item in the returned tuple will be *preserve* unchanged.

pvl.lexer.**lex_singlechar_comments**(*char: str*, *lexeme: str*, *preserve: dict*, *comments: dict) -> (<class 'str'>, <class 'dict'>)*

Returns a modified *lexeme* string and a modified *preserve* dict in a two-tuple.

This is a lexer() helper function for determining how to modify *lexeme* and *preserve* based on the single character in *char* which may or may not be a comment character.

If the *preserve* 'state' value is Preserve.COMMENT then the value of lex_preserve() is returned.

If *char* is among the keys of the *comments* dict, then the returned *lexeme* will be the concatenation of *lexeme* and *char*. returned *preserve* dict will have its 'state' value set to Preserve.COMMENT and its 'end' value set to the value of *comments[char]*.

Otherwise return *lexeme* and *preserve* unchanged in the two-tuple.

pvl.lexer.**lexer**(*s: str*, *g=<pvl.grammar.PVLGrammar object>*, *d=<pvl.decoder.PVLDecoder object>*)

This is a generator function that returns pvl.Token objects based on the passed in string, *s*, when the generator's next() is called.

A call to send(*t*) will 'return' the value *t* to the generator, which will be yielded upon calling next(). This allows a user to 'peek' at the next token, but return it if they don't like what they see.

g is expected to be an instance of pvl.grammar, and *d* an instance of pvl.decoder. The lexer will perform differently, given different values of *g* and *d*.

7.1.8 pvl.new module

7.1.9 pvl.parser module

Parameter Value Language parser.

The definition of PVL used in this module is based on the Consultive Committee for Space Data Systems, and their Parameter Value Language Specification (CCSD0006 and CCSD0008), CCSDS 6441.0-B-2, referred to as the Blue Book with a date of June 2000.

Some of the documention in this module represents the structure diagrams from the Blue Book for parsing PVL in a Backus–Naur form.

So Figure 1-1 from the Blue Book would be represented as :

<Item-A> ::= ([<Item-B>+ | <Item-C>] <Item-D>)*

Finally, the Blue Book defines <WSC> as a possibly empty collection of white space characters or comments:

<WSC> ::= (<white-space-character> | <comment>)*

However, to help remember that <WSC> could be empty, we will typically always show it as <WSC>*.

Likewise the <Statement-Delimiter> is defined as:

<Statement-Delimiter> ::= <WSC>* [';' | <EOF>]

However, since all elements are optional, we will typically show it as [<Statement-Delimiter>].

The parser deals with managing the tokens that come out of the lexer. Once the parser gets to a state where it has something that needs to be converted to a Python object and returned, it uses the decoder to make that conversion.

Throughout this module, various parser functions will take a *tokens: collections.abc.Generator* parameter. In all cases, *tokens* is expected to be a *generator iterator* which provides `pvl.token.Token` objects. It should allow for a generated object to be 'returned' via the generator's send() function. When parsing the first object from *tokens*, if an unexpected object is encountered, it will 'return' the object to *tokens*, and raise a `ValueError`, so that `try-except` blocks can be used, and the *generator iterator* is left in a good state. However, if a parsing anomaly is discovered deeper in parsing a PVL sequence, then a `ValueError` will be thrown into the *tokens* generator iterator (via .throw()).

class pvl.parser.**EmptyValueAtLine**

Bases: `str`

Empty string to be used as a placeholder for a parameter without a value.

When a label contains a parameter without a value, it is normally considered a broken label in PVL. To allow parsing to continue, we can rectify the broken parameter-value pair by setting the value to have a value of EmptyValueAtLine, which is an empty string (and can be treated as such) with some additional properties.

The argument *lineno* should be the line number of the error from the original document, which will be available as a property.

Examples::

```
>>> from pvl.parser import EmptyValueAtLine
>>> EV1 = EmptyValueAtLine(1)
>>> EV1
EmptyValueAtLine(1 does not have a value. Treat as an empty string)
>>> EV1.lineno
1
>>> print(EV1)
<BLANKLINE>
```

```
>>> EV1 + 'foo'
'foo'
>>> # Can be turned into an integer and float as 0:
>>> int(EV1)
0
>>> float(EV1)
0.0
```

class pvl.parser.**ODLParser**(*grammar=None*, *decoder=None*, *lexer_fn=None*, *module_class=<class 'pvl.collections.PVLModule'>*, *group_class=<class 'pvl.collections.PVLGroup'>*, *object_class=<class 'pvl.collections.PVLObject'>*)

Bases: *pvl.parser.PVLParser*

A parser based on the rules in the PDS3 Standards Reference (version 3.8, 27 Feb 2009) Chapter 12: Object Description Language Specification and Usage.

It extends PVLParser.

parse_set(*tokens: collections.abc.Generator*) → set

Overrides the parent function to return the decoded <Set> as a Python `set`.

The ODL specification only allows scalar_values in Sets, since ODL Sets cannot contain other ODL Sets, an ODL Set can be represented as a Python `set` (unlike PVL Sets, which must be represented as a Python `frozenset` objects).

parse_units(*value*, *tokens: collections.abc.Generator*) → str

Extends the parent function, since ODL only allows units on numeric values, any others will result in a ValueError.

class pvl.parser.**OmniParser**(*grammar=None*, *decoder=None*, *lexer_fn=None*, *module_class=<class 'pvl.collections.PVLModule'>*, *group_class=<class 'pvl.collections.PVLGroup'>*, *object_class=<class 'pvl.collections.PVLObject'>*)

Bases: *pvl.parser.PVLParser*

A permissive PVL/ODL/ISIS label parser that attempts to parse all forms of "PVL" that are thrown at it.

parse(*s: str*)

Extends the parent function.

If *any* line ends with a dash (-) followed by a carriage return, form-feed, or newline, plus one or more whitespace characters on the following line, then those characters, and all whitespace characters that begin the next line will be removed.

parse_assignment_statement(*tokens: collections.abc.Generator*) → tuple

Extends the parent function to allow for more permissive parsing. If an Assignment-Statement is blank, then the value will be assigned an EmptyValueAtLine object.

parse_module_post_hook(*module: pvl.collections.MutableMappingSequence*, *tokens: collections.abc.Generator*)

Overrides the parent function to allow for more permissive parsing. If an Assignment-Statement is blank, then the value will be assigned an EmptyValueAtLine object.

parse_value_post_hook(*tokens: collections.abc.Generator*)

Overrides the parent function to allow for more permissive parsing.

If the next token is a reserved word or delimiter, then it is returned to the *tokens* and an EmptyValueAtLine object is returned as the value.

class pvl.parser.**PVLParser**(*grammar=None*, *decoder=None*, *lexer_fn=None*, *module_class=<class 'pvl.collections.PVLModule'>*, *group_class=<class 'pvl.collections.PVLGroup'>*, *object_class=<class 'pvl.collections.PVLObject'>*)

Bases: `object`

A parser based on the rules in the CCSDS-641.0-B-2 'Blue Book' which defines the PVL language.

Parameters

- **grammar** – A pvl.grammar object, if None or not specified, it will be set to the grammar parameter of *decoder* (if *decoder* is not None) or will default to *pvl.grammar.OmniGrammar()*.
- **decoder** – defaults to *pvl.decoder.OmniDecoder()*.
- **lexer_fn** – must be a lexer function that takes a `str`, a grammar, and a decoder, as *pvl.lexer.lexer()* does, which is the default if none is given.
- **module_class** – must be a subclass of PVLModule, and is the type of object that will be returned from this parser's *parse()* function.
- **group_class** – must be a subclass of PVLGroup, and is the type that will be used to hold PVL elements when a PVL Group is encountered during parsing, and must be able to be added to via an `.append()` function which should take a two-tuple of name and value.
- **object_class** – must be a subclass of PVLObject, and is the type that will be used to hold PVL elements when a PVL Object is encountered during parsing, otherwise similar to *group_class*.

aggregation_cls(*begin: str*)
Returns an initiated object of the group_class or object_class as specified on this parser's creation, according to the value of *begin*. If *begin* does not match the Group or Object keywords for this parser's grammar, then it will raise a ValueError.

parse(*s: str*)
Converts the string, *s* to a PVLModule.

static parse_WSC_until(*token: str*, *tokens: collections.abc.Generator*) → bool
Consumes objects from *tokens*, if the object's *.is_WSC()* function returns *True*, it will continue until *token* is encountered and will return *True*. If it encounters an object that does not meet these conditions, it will 'return' that object to *tokens* and will return *False*.

tokens is expected to be a *generator iterator* which provides `pvl.token` objects.

parse_aggregation_block(*tokens: collections.abc.Generator*)
Parses the tokens for an Aggregation Block, and returns the modcls object that is the result of the parsing and decoding.

<Aggregation-Block> ::= <Begin-Aggegation-Statement> (<WSC>* (Assignment-Statement | Aggregation-Block) <WSC>*)+ <End-Aggregation-Statement>

The Begin-Aggregation-Statement Name must match the Block-Name in the paired End-Aggregation-Statement if a Block-Name is present in the End-Aggregation-Statement.

parse_around_equals(*tokens: collections.abc.Generator*) → None
Parses white space and comments on either side of an equals sign.

tokens is expected to be a *generator iterator* which provides `pvl.token` objects.

This is shared functionality for Begin Aggregation Statements and Assignment Statements. It basically covers parsing anything that has a syntax diagram like this:

<WSC>* '=' <WSC>*

parse_assignment_statement(*tokens: collections.abc.Generator*) → tuple
Parses the tokens for an Assignment Statement.

The returned two-tuple contains the Parameter Name in the first element, and the Value in the second.

<Assignment-Statement> ::= <Parameter-Name> <WSC>* '=' <WSC>* <Value> [<Statement-Delimiter>]

parse_begin_aggregation_statement(*tokens: collections.abc.Generator*) → tuple
Parses the tokens for a Begin Aggregation Statement, and returns the name Block Name as a `str`.

<Begin-Aggregation-Statement-block> ::= <Begin-Aggegation-Statement> <WSC>* '=' <WSC>* <Block-Name> [<Statement-Delimiter>]

Where <Block-Name> ::= <Parameter-Name>

parse_end_aggregation(*begin_agg: str*, *block_name: str*, *tokens: collections.abc.Generator*) → None
Parses the tokens for an End Aggregation Statement.

<End-Aggregation-Statement-block> ::= <End-Aggegation-Statement> [<WSC>* '=' <WSC>* <Block-Name>] [<Statement-Delimiter>]

Where <Block-Name> ::= <Parameter-Name>

parse_end_statement(*tokens: collections.abc.Generator*) → None
Parses the tokens for an End Statement.

<End-Statement> ::= "END" (<WSC>* | [<Statement-Delimiter>])

parse_module(*tokens: collections.abc.Generator*)
Parses the tokens for a PVL Module.

<PVL-Module-Contents> ::= (<Assignment-Statement> | <WSC>* | <Aggregation-Block>)* [<End-Statement>]

parse_module_post_hook(*module: pvl.collections.MutableMappingSequence*, *tokens: collections.abc.Generator*)
This function is meant to be overridden by subclasses that may want to perform some extra processing if 'normal' parse_module() operations fail to complete. See OmniParser for an example.

This function shall return a two-tuple, with the first item being the *module* (altered by processing or unaltered), and the second item being a boolean that will signal whether the tokens should continue to be parsed to accumulate more elements into the returned *module*, or whether the *module* is in a good state and should be returned by parse_module().

If the operations within this function are unsuccessful, it should raise an exception (any exception descended from Exception), which will result in the operation of parse_module() as if it were not overridden.

parse_sequence(*tokens: collections.abc.Generator*) → list
Parses a PVL Sequence.

<Set> ::= "(" <WSC>* [<Value> <WSC>* ("," <WSC>* <Value> <WSC>*)*] ")"

Returns the decoded <Sequence> as a Python `list`.

parse_set(*tokens: collections.abc.Generator*) → frozenset
Parses a PVL Set.

<Set> ::= "{" <WSC>* [<Value> <WSC>* ("," <WSC>* <Value> <WSC>*)*] "}"

Returns the decoded <Set> as a Python `frozenset`. The PVL specification doesn't seem to indicate that a PVL Set has distinct values (like a Python `set`), only that the ordering of the values is unimportant. For now, we will implement PVL Sets as Python `frozenset` objects.

They are returned as `frozenset` objects because PVL Sets can contain as their elements other PVL Sets, but since Python `set` objects are non-hashable, they cannot be members of a set, however, `frozenset` objects can.

static parse_statement_delimiter(*tokens: collections.abc.Generator*) → bool
Parses the tokens for a Statement Delimiter.

tokens is expected to be a *generator iterator* which provides `pvl.token` objects.

<Statement-Delimiter> ::= <WSC>* (<white-space-character> | <comment> | ';' | <EOF>)

Although the above structure comes from Figure 2-4 of the Blue Book, the <white-space-character> and <comment> elements are redundant with the presence of [WSC]* so it can be simplified to:

<Statement-Delimiter> ::= <WSC>* [';' | <EOF>]

Typically written [<Statement-Delimiter>].

parse_units(*value*, *tokens: collections.abc.Generator*) → str
Parses PVL Units Expression.

<Units-Expression> ::= "<" [<white-space>] <Units-Value> [<white-space>] ">"

and

<Units-Value> ::= <units-character>

[[<units-character> | <white-space>]* <units-character>]

Returns the *value* and the <Units-Value> as a `Units()` object.

parse_value(*tokens: collections.abc.Generator*)
Parses PVL Values.

<Value> ::= (<Simple-Value> | <Set> | <Sequence>) [<WSC>* <Units Expression>]

Returns the decoded <Value> as an appropriate Python object.

parse_value_post_hook(*tokens*)
This function is meant to be overridden by subclasses that may want to perform some extra processing if 'normal' parse_value() operations fail to yield a value. See OmniParser for an example.

This function shall return an appropriate Python value, similar to what parse_value() would return.

If the operations within this function are unsuccessful, it should raise a ValueError which will result in the operation of parse_value() as if it were not overridden.

7.1.10 pvl.pvl_translate module

A program for converting PVL text to a specific PVL dialect.

The `pvl_translate` program will read a file with PVL text (any of the kinds of files that *pvl.load()* reads) or STDIN and will convert that PVL text to a particular PVL dialect. It is not particularly robust, and if it cannot make simple conversions, it will raise errors.

class pvl.pvl_translate.**JSONWriter**
Bases: *pvl.pvl_translate.Writer*

dump(*dictlike: dict*, *outfile: os.PathLike*)

class pvl.pvl_translate.**PVLWriter**(*encoder*)
Bases: *pvl.pvl_translate.Writer*

dump(*dictlike: dict*, *outfile: os.PathLike*)

class pvl.pvl_translate.**Writer**
Bases: object

Base class for writers. Descendents must implement dump().

dump(*dictlike: dict*, *outfile: os.PathLike*)

pvl.pvl_translate.**arg_parser**(*formats*)

pvl.pvl_translate.**main**(*argv=None*)

7.1.11 pvl.pvl_validate module

A program for testing and validating PVL text.

The `pvl_validate` program will read a file with PVL text (any of the kinds of files that `pvl.load()` reads) and will report on which of the various PVL dialects were able to load that PVL text, and then also reports on whether the `pvl` library can encode the Python Objects back out to PVL text.

You can imagine some PVL text that could be loaded, but is not able to be written out in a particular strict PVL dialect (like PDS3 labels).

`pvl.pvl_validate.`**`arg_parser`**`()`

`pvl.pvl_validate.`**`build_line`**`(`*elements: list*, *widths: list*, *sep=' '*`)` → str
Returns a string formatted from the *elements* and *widths* provided.

`pvl.pvl_validate.`**`main`**`(`*argv=None*`)`

`pvl.pvl_validate.`**`pvl_flavor`**`(`*text*, *dialect*, *decenc: dict*, *filename*, *verbose=False) -> (<class 'bool'>*, *<class 'bool'>*`)`
Returns a two-tuple of booleans which indicate whether the *text* could be loaded and then encoded.

The first boolean in the two-tuple indicates whether the *text* could be loaded with the given parser, grammar, and decoder. The second indicates whether the loaded PVL object could be encoded with the given encoder, grammar, and decoder. If the first element is False, the second will be None.

`pvl.pvl_validate.`**`report`**`(`*reports: list*, *flavors: list*`)` → str
Returns a multi-line string which is the pretty-printed report given the list of *reports*.

`pvl.pvl_validate.`**`report_many`**`(`*r_list: list*, *flavors: list*`)` → str
Returns a multi-line, table-like string which is the pretty-printed report of the items in *r_list*.

7.1.12 pvl.token module

`class` `pvl.token.`**`Token`**`(`*content*, *grammar=None*, *decoder=None*, *pos=0*`)`
Bases: `str`

A PVL-aware string.

Variables

- **`content`** – A string that is the Token text.
- ***`grammar`*** – A pvl.grammar object, if None or not specified, it will be set to the grammar parameter of *decoder* (if *decoder* is not None) or will default to PVLGrammar().
- ***`decoder`*** – A pvl.decoder object, defaults to PVLDecoder(grammar=*grammar*).
- **`pos`** – Integer that describes the starting position of this Token in the source string, defaults to zero.

`is_WSC``()` → bool
Return true if the Token is white space characters or comments according to the Token's grammar, false otherwise.

`is_begin_aggregation``()` → bool
Return true if the Token is a begin aggregation keyword (e.g. 'BEGIN_GROUP' in PVL) according to the Token's grammar, false otherwise.

`is_comment``()` → bool
Return true if the Token is a comment according to the Token's grammar (defined as beginning and ending with comment delimieters), false otherwise.

is_datetime() → bool
Return true if the Token's decoder can convert the Token to a datetime, false otherwise.

Separate is_date() or is_time() functions aren't needed, since PVL parsing doesn't distinguish between them. If a user needs that distinction the decoder's decode_datetime(self) function should return a datetime time, date, or datetime object, as appropriate, and a user can use isinstance() to check.

is_decimal() → bool
Return true if the Token's decoder can convert the Token to a decimal value, false otherwise.

is_delimiter() → bool
Return true if the Token is a delimiter character (e.g. the ';' in PVL) according to the Token's grammar, false otherwise.

is_end_statement() → bool
Return true if the Token matches an end statement from its grammar, false otherwise.

is_non_decimal() → bool
Return true if the Token's decoder can convert the Token to a numeric non-decimal value, false otherwise.

is_numeric() → bool
Return true if the Token's is_decimal() or is_non_decimal() functions return true, false otherwise.

is_parameter_name() → bool
Return true if the Token is an unquoted string that isn't a reserved_keyword according to the Token's grammar, false otherwise.

is_quote() → bool
Return true if the Token is a quote character according to the Token's grammar, false otherwise.

is_quoted_string() → bool
Return true if the Token can be converted to a quoted string by the Token's decoder, false otherwise.

is_simple_value() → bool
Return true if the Token's decoder can convert the Token to a 'simple value', however the decoder defines that, false otherwise.

is_space() → bool
Return true if the Token contains whitespace according to the definition of whitespace in the Token's grammar and there is at least one character, false otherwise.

is_string() → bool
Return true if either the Token's is_quoted_string() or is_unquoted_string() return true, false otherwise.

is_unquoted_string() → bool
Return false if the Token has any reserved characters, comment characters, whitespace characters or could be interpreted as a number, date, or time according to the Token's grammar, true otherwise.

isnumeric() → bool
Overrides `str.isnumeric()` to be the same as Token's is_numeric() function, so that we don't get inconsisent behavior if someone forgets an underbar.

isspace() → bool
Overrides `str.isspace()` to be the same as Token's is_space() function, so that we don't get inconsisent behavior if someone forgets an underbar.

lstrip(*chars=None*)
Extends `str.lstrip()` to strip whitespace according to the definition of whitespace in the Token's grammar instead of the default Python whitespace definition.

replace(**args*)
Extends `str.replace()` to return a Token.

rstrip(*chars=None*)
Extends `str.rstrip()` to strip whitespace according to the definition of whitespace in the Token's grammar instead of the default Python whitespace definition.

split(*sep=None*, *maxsplit=-1*) → list
Extends `str.split()` that calling split() on a Token returns a list of Tokens.

strip(*chars=None*)
Extends `str.strip()` to strip whitespace according to the definition of whitespace in the Token's grammar instead of the default Python whitespace definition.

7.1.13 Module contents

Python implementation of PVL (Parameter Value Language).

`pvl.`**load**(*path*, *parser=None*, *grammar=None*, *decoder=None*, *encoding=None*, ***kwargs*)
Returns a Python object from parsing the file at *path*.

Parameters

- **path** – an `os.PathLike` which presumably has a PVL Module in it to parse.
- **parser** – defaults to *pvl.parser.OmniParser()*.
- **grammar** – defaults to *pvl.grammar.OmniGrammar()*.
- **decoder** – defaults to *pvl.decoder.OmniDecoder()*.
- **encoding** – defaults to None, and has the same meaning as for `open()`.
- ****kwargs** – the keyword arguments that will be passed to *loads()* and are described there.

If *path* is not an `os.PathLike`, it will be assumed to be an already-opened file object, and `.read()` will be applied to extract the text.

If the `os.PathLike` or file object contains some bytes decodable as text, followed by some that is not (e.g. an ISIS cube file), that's fine, this function will just extract the decodable text.

`pvl.`**loads**(*s: str*, *parser=None*, *grammar=None*, *decoder=None*, ***kwargs*)
Deserialize the string, *s*, as a Python object.

Parameters

- **s** – contains some PVL to parse.
- **parser** – defaults to *pvl.parser.OmniParser()*.
- **grammar** – defaults to *pvl.grammar.OmniGrammar()*.
- **decoder** – defaults to *pvl.decoder.OmniDecoder()*.
- ****kwargs** – the keyword arguments to pass to the *parser* class if *parser* is none.

`pvl.`**dump**(*module*, *path*, ***kwargs*)
Serialize *module* as PVL text to the provided *path*.

Parameters

- **module** – a `PVLModule` or `dict`-like object to serialize.
- **path** – an `os.PathLike`
- ****kwargs** – the keyword arguments to pass to *dumps()*.

If *path* is an os.PathLike, it will attempt to be opened and the serialized module will be written into that file via the pathlib.Path.write_text() function, and will return what that function returns.

If *path* is not an os.PathLike, it will be assumed to be an already-opened file object, and .write() will be applied on that object to write the serialized module, and will return what that function returns.

pvl.**dumps**(*module*, *encoder=None*, *grammar=None*, *decoder=None*, ***kwargs*) → str
Returns a string where the *module* object has been serialized to PVL syntax.

Parameters

- **module** – a PVLModule or dict like object to serialize.
- **encoder** – defaults to pvl.parser.PDSLabelEncoder().
- **grammar** – defaults to *pvl.grammar.ODLGrammar()*.
- **decoder** – defaults to *pvl.decoder.ODLDecoder()*.
- ****kwargs** – the keyword arguments to pass to the encoder class if *encoder* is none.

class pvl.**PVLModule**(**args*, ***kwargs*)
Bases: *pvl.collections.OrderedMultiDict*

class pvl.**PVLGroup**(**args*, ***kwargs*)
Bases: *pvl.collections.PVLAggregation*

class pvl.**PVLObject**(**args*, ***kwargs*)
Bases: *pvl.collections.PVLAggregation*

class pvl.**Quantity**
Bases: *pvl.collections.Quantity*

A simple collections.namedtuple object to contain a value and units parameter.

If you need more comprehensive units handling, you may want to use the astropy.units.Quantity object, the pint.Quantity object, or some other 3rd party object. Please see the documentation on *Quantities: Values and Units* for how to use 3rd party Quantity objects with pvl.

class pvl.**Units**
Bases: *pvl.collections.Quantity*

CHAPTER 8

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

8.1 Types of Contributions

8.1.1 Report Bugs or Ask for Features via Issues

We want to hear from you! You can report bugs, ask for new features, or just raise issues or concerns via logging an Issue via our GitHub repo.

8.1.2 Fix Bugs or Implement Features

Look through the GitHub Issues for bugs to fix or feautures to implement. If anything looks tractable to you, work on it. Most (if not all) PRs should be based on an Issue, so if you're thinking about doing some coding on a topic that isn't covered in an Issue, please author one so you can get some feedback while you work on your PR.

8.1.3 Write Documentation

pvl could always use more documentation, whether as part of the official pvl docs, in docstrings, or even on the web in blog posts, articles, and such.

8.1.4 Submit Feedback

The best way to send feedback is to file an Issue.

8.2 Get Started!

Ready to contribute? Here's how to set up *pvl* for local development.

1. Fork the *pvl* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/pvl.git
```

3. Install your local copy into a virtual environment like virtualenv or conda. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv pvl
$ cd pvl/
$ pip install -r requirements.txt
```

   If you are a conda user:

```
$ cd pvl/
$ conda env create -n pvldev -f environment.yml
$ conda activate pvldev
$ pip install --no-deps -e .
```

   The last *pip install* installs pvl in "editable" mode which facilitates testing.

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ make lint
$ make test
$ make test-all
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request.

8.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in HISTORY.rst and potentially update the README.rst or other documentation files.

3. The pull request should work for Python 3.6, 3.7, and 3.8. Check https://travis-ci.org/github/planetarypy/pvl and make sure that the tests pass for all supported Python versions.

8.4 Tips

To run a subset of tests:

```
$ pytest tests/test_pvl.py
```

8.5 What to expect

We want to keep development moving forward, and you should expect activity on your PR within a week or so.

8.6 Rules for Merging Pull Requests

Any change to resources in this repository must be through pull requests (PRs). This applies to all changes to documentation, code, binary files, etc. Even long term committers must use pull requests.

In general, the submitter of a PR is responsible for making changes to the PR. Any changes to the PR can be suggested by others in the PR thread (or via PRs to the PR), but changes to the primary PR should be made by the PR author (unless they indicate otherwise in their comments). In order to merge a PR, it must satisfy these conditions:

1. Have been open for 24 hours.
2. Have one approval.
3. If the PR has been open for 1 week without approval or comment, then it may be merged without any approvals.

Pull requests should sit for at least 24 hours to ensure that contributors in other timezones have time to review. Consideration should also be given to weekends and other holiday periods to ensure active committers all have reasonable time to become involved in the discussion and review process if they wish.

In order to encourage involvement and review, we encourage at least one explicit approval from committers that are not the PR author.

However, in order to keep development moving along with our low number of active contributors, if a PR has been open for a week without comment, then it could be committed without an approval.

The default for each contribution is that it is accepted once no committer has an objection, and the above requirements are satisfied.

In the case of an objection being raised in a pull request by another committer, all involved committers should seek to arrive at a consensus by way of addressing concerns being expressed by discussion, compromise on the proposed change, or withdrawal of the proposed change.

If a contribution is controversial and committers cannot agree about how to get it merged or if it should merge, then the developers will escalate the matter to the PlanetaryPy TC for guidance. It is expected that only a small minority of issues be brought to the PlanetaryPy TC for resolution and that discussion and compromise among committers be the default resolution mechanism.

Exceptions to the above are minor typo fixes or cosmetic changes that don't alter the meaning of a document. Those edits can be made via a PR and the requirement for being open 24 h is waived in this case.

8.7 PVL People

- A PVL **Contributor** is any individual creating or commenting on an issue or pull request. Anyone who has authored a PR that was merged should be listed in the AUTHORS.rst file.
- A PVL **Committer** is a subset of contributors who have been given write access to the repository.

All contributors who get a non-trivial contribution merged can become Committers. Individuals who wish to be considered for commit-access may create an Issue or contact an existing Committer directly.

Committers are expected to follow this policy and continue to send pull requests, go through proper review, etc.

CHAPTER 9

Credits

The pvl library was originally developed by Trevor Olson.

9.1 Authors

- Trevor Olson <trevor@heytrevor.com> (Original Author and former development lead)
- Sarah Braden <braden.sarah@gmail.com>
- Michael Aye <kmichael.aye@gmail.com>
- Austin Godber <godber@uberhip.com>
- Perry Vargas <perrybvargas@gmail.com>
- Benoit Seignovert
- Ross Beyer

9.2 Acknowledgements

Thanks to Michael Aye, Andrew Annex, and Chase Million for valuable discussions during the lead-up to the 1.0.0 release.

CHAPTER 10

History

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog, and this project adheres to Semantic Versioning.

When updating this file, please add an entry for your change under *Not Yet Released* and one of the following headings:

- Added - for new features.
- Changed - for changes in existing functionality.
- Deprecated - for soon-to-be removed features.
- Removed - for now removed features.
- Fixed - for any bug fixes.
- Security - in case of vulnerabilities.

If the heading does not yet exist under *Not Yet Released*, then add it as a 3rd level heading, underlined with pluses (see examples below).

When preparing for a public release add a new 2nd level heading, underlined with dashes under *Not Yet Released* with the version number and the release date, in year-month-day format (see examples below).

10.1 Not Yet Released

10.2 1.3.2 (2022-02-05)

10.2.1 Fixed

- The parser was requesting the next token after an end-statement, even though nothing was done with this token (in the future it could be a comment that should be processed). In the very rare case where all of the "data" bytes in a file with an attached PVL label (like a .IMG or .cub file) actually convert to UTF with no whitespace

characters, that next token will take an unacceptable amount of time to return, if it does at all. The parser now does not request additional tokens once an end-statement is identified (Issue 104).

10.3 1.3.1 (2022-02-05)

10.3.1 Fixed

- Deeply nested Aggregation Blocks (Object or Group) which had mis-matched Block Names should now properly result in LexerErrors instead of resulting in StopIteration Exceptions (Issue 100).
- The default "Omni" parsing strategy, now considers the ASCII NULL character ("0") a "reserved character." The practical effect is that the ASCII NULL can not be in parameter names or unquoted strings (but would still be successfully parsed in quoted strings). This means that PVL-text that might have incorrectly used ASCII NULLs as delimiters will once again be consumed by our omnivorous parser (Issue 98).

10.4 1.3.0 (2021-09-10)

10.4.1 Added

- pvl.collections.Quantity objects now have __int__() and __float__() functions that will return the int and float versions of their .value parameter to facilitate numeric operations with Quantity objects (Issue 91).
- pvl.load() now has an *encoding*= parameter that is identical in usage to the parameter passed to *open()*, and will attempt to decode the whole file as if it had been encoded thusly. If it encounters a decoding error, it will fall back to decoding the bytes one at a time as ASCII text (Issue 93).

10.4.2 Fixed

- If the PVL-text contained characters beyond the set allowed by the PVL specification, the OmniGrammar would refuse to parse them. This has been fixed to allow any valid character to be parsed, so that if there are weird UTF characters in the PVL-text, you'll get those weird UTF characters in the returned dict-like. When the stricter PVL, ODL, or PDS3 dialects are used to "load" PVL-text, they will properly fail to parse this text (Issue 93).
- Empty parameters inside groups or objects (but not at the end), would cause the default "Omni" parsing strategy to go into an infinite loop. Empty parameters in PVL, ODL, and PDS3 continue to not be allowed (Issue 95).

10.5 1.2.1 (2021-05-31)

10.5.1 Added

- So many tests, increased coverage by about 10%.

10.5.2 Fixed

- Attempting to import *pvl.new* without *multidict* being available, will now properly yield an ImportError.
- The *dump()* and *dumps()* functions now properly overwritten in *pvl.new*.

- All encoders that descended from PVLEncoder didn't properly have group_class and object_class arguments to their constructors, now they do.
- The *char_allowed()* function in grammar objects now raises a more useful ValueError than just a generic Exception.
- The new *collections.PVLMultiDict* wasn't correctly inserting Mapping objects with the *insert_before()* and *insert_after()* methods.
- The *token.Token* class's *__index__()* function didn't always properly return an index.
- The *token.Token* class's *__float__()* function would return int objects if the token could be converted to int. Now always returns floats.

10.6 1.2.0 (2021-03-27)

10.6.1 Added

- Added a default_timezone parameter to grammar objects so that they could both communicate whether they had a default timezone (if not None), and what it was.
- Added a pvl.grammar.PDSGrammar class that specifies the default UTC time offset.
- Added a pvl.decoder.PDSLabelDecoder class that properly enforces only milisecond time precision (not microsecond as ODL allows), and does not allow times with a +HH:MM timezone specifier. It does assume any time without a timezone specifier is a UTC time.
- Added a `real_cls` parameter to the decoder classes, so that users can specify an arbitrary type with which real numbers in the PVL-text could be returned in the dict-like from the loaders (defaults to `float` as you'd expect).
- The encoders now support a broader range of real types to complement the decoders.

10.6.2 Changed

- Improved some build and test functionality.
- Moved the is_identifier() static function from the ODLEncoder to the ODLDecoder where it probably should have always been.

10.6.3 Fixed

- Very long Python `str` objects that otherwise qualified as ODL/PDS3 Symbol Strings, would get written out with single-quotes, but they would then be split across lines via the formatter, so they should be written as Text Strings with double-quotes. Better protections have been put in place.
- pvl.decoder.ODLDecoder now will return both "aware" and "naive" datetime objects (as appropriate) since "local" times without a timezone are allowed under ODL.
- pvl.decoder.ODLDecoder will now properly reject any unquoted string that does not parse as an ODL Identifier.
- pvl.decoder.ODLDecoder will raise an exception if there is a seconds value of 60 (which the PVLDecoder allows)
- pvl.encoder.ODLEncoder will raise an exception if given a "naive" time object.

- pvl.encoder.PDSLabelEncoder will now properly raise an exception if a time or datetime object cannot be represented with only milisecond precision.

10.7 1.1.0 (2020-12-04)

10.7.1 Added

- Modified *pvl_validate* to more robustly deal with errors, and also provide more error-reporting via *-v* and *-vv*.
- Modified ISISGrammar so that it can parse comments that begin with an octothorpe (#).

10.7.2 Fixed

- Altered documentation in grammar.py that was incorrectly indicating that there were parameters that could be passed on object initiation that would alter how those objects behaved.

10.8 1.0.1 (2020-09-21)

10.8.1 Fixed

- The PDSLabelEncoder was improperly raising an exception if the Python datetime object to encode had a tzinfo component that had zero offset from UTC.

10.9 1.0.0 (2020-08-23)

This production version of the pvl library consists of significant API and functionality changes from the 0.x version that has been in use for 5 years (a credit to Trevor Olson's skills). The documentation has been significantly upgraded, and various granular changes over the 10 alpha versions of 1.0.0 over the last 8 months are detailed in their entries below. However, here is a high-level overview of what changed from the 0.x version:

10.9.1 Added

- `pvl.load()` and `pvl.dump()` take all of the arguments that they could take before (string containing a filename, byte streams, etc.), but now also accept any `os.PathLike` object, or even an already-opened file object.
- `pvl.loadu()` function will load PVL text from URLs.
- Utility programs *pvl_validate* and *pvl_translate* were added, please see the "Utility Programs" section of the documentation for more information.
- The library can now parse and encode PVL Values with Units expressions with third-party quantity objects like *astropy.units.Quantity* and *pint.Quantity*. Please see the "Quantities: Values and Units" section of the documentation.
- Implemented a new PVLMultiDict (optional, needs 3rd party multidict library) which which has more pythonic behaviors than the existing OrderedMultiDict. Experiment with getting it returned by the loaders by altering your import statement to `import pvl.new as pvl` and then using the loaders as usual to get the new object returned to you.

10.9.2 Changed

- Only guaranteed to work with Python 3.6 and above.
- Rigorously implemented the three dialects of PVL text: PVL itself, ODL, and the PDS3 Label Standard. There is a fourth de-facto dialect, that of ISIS cube labels that is also handled. Please see the "Standards & Specifications" section of the documentation.
- There is now a default dialect for the dump functions: the PDS3 Label Standard. This is different and more strict than before, but other output dialects are possible. Please see the "Writing out PVL text" section in the documentation for more information, and how to enable an output similar to the 0.x output.
- There are now `pvl.collections` and `pvl.exceptions` modules. There was previously an internal `pvl._collections` module, and the exception classes were scattered through the other modules.

10.9.3 Fixed

- All `datetime.time` and `datetime.datetime` objects returned from the loaders are now timezone "aware." Previously some were and some were not.
- Functionality to correctly parse dash (-) continuation lines in ISIS output is now supported.
- The library now properly parses quoted strings that include backslashes.

10.9.4 Deprecated

- The *pvl.collections.Units* object is deprecated in favor of the new `pvl.collections.Quantity` object (really a name-only change, no functionality difference).

10.10 1.0.0-alpha.9 (2020-08-18)

- Minor addition to pvl.collections.MutableMappingSequence.
- Implemented PVLMultiDict which is based on the 3rd Party *multidict.MultiDict* object as an option to use instead of the default OrderedMultiDict. The new PVLMultiDict is better aligned with the Python 3 way that Mapping objects behave.
- Enhanced the existing OrderedMultiDict with some functionality that extends its behavior closer to the Python 3 ideal, and inserted warnings about how the retained non-Python-3 behaviors might be removed at the next major patch.
- Implemented pvl.new that can be included for those that wish to try out what getting the new PVLMultiDict returned from the loaders might be like by just changing an import statement.

10.11 1.0.0-alpha.8 (2020-08-01)

- Renamed the _collections module to just collections.
- Renamed the Units class to Quantity (Units remains, but has a deprecation warning).
- Defined a new ABC: pvl.collections.MutableMappingSequence
- More detail for these changes can be found in Issue #62.

10.12 1.0.0-alpha.7 (2020-07-29)

- Created a new exceptions.py module and grouped all pvl Exceptions there. Addresses #58
- Altered the message that LexerError emits to provide context around the character that caused the error.
- Added bump2version configuration file.

10.13 1.0.0-alpha.6 (2020-07-27)

- Enforced that all datetime.time and datetime.datetime objects returned should be timezone "aware." This breaks 0.x functionality where some were and some weren't. Addresses #57.

10.14 1.0.0-alpha.5 (2020-05-30)

- ISIS creates PVL text with unquoted plus signs ("+"), needed to adjust the ISISGrammar and OmniGrammar objects to parse this properly (#59).
- In the process of doing so, realized that we have some classes that optionally take a grammar and a decoder, and if they aren't given, to default. However, a decoder *has* a grammar object, so if a grammar isn't provided, but a decoder is, the grammar should be taken from the decoder, otherwise you could get confusing behavior.
- Updated pvl_validate to be explicit about these arguments.
- Added a –version argument to both pvl_translate and pvl_validate.

10.15 1.0.0.-alpha.4 (2020-05-29)

- Added the pvl.loadu() function as a convenience function to load PVL text from URLs.

10.16 1.0.0-alpha.3 (2020-05-28)

- Implemented tests in tox and Travis for Python 3.8, and discovered a bug that we fixed (#54).

10.17 1.0.0-alpha.2 (2020-04-18)

- The ability to deal with 3rd-party 'quantity' objects like astropy.units.Quantity and pint.Quantity was added and documented, addresses #22.

10.18 1.0.0-alpha.1 (2020-04-17)

This is a bugfix on 1.0.0-alpha to properly parse scientific notation and deal with properly catching an error.

10.19 1.0.0-alpha (winter 2019-2020)

This is the alpha version of release 1.0.0 for pvl, and the items here and in other 'alpha' entries may be consolidated when 1.0.0 is released. This work is categorized as 1.0.0-alpha because backwards-incompatible changes are being introduced to the codebase.

- Refactored code so that it will no longer support Python 2, and is only guaranteed to work with Python 3.6 and above.
- Rigorously implemented the three dialects of PVL text: PVL itself, ODL, and the PDS3 Label Standard. There is a fourth de-facto dialect, that of ISIS cube labels that is also handled. These dialects each have their own grammars, parsers, decoders, and encoders, and there are also some 'Omni' versions of same that handle the widest possible range of PVL text.
- When parsing via the loaders, `pvl` continues to consume as wide a variety of PVL text as is reasonably possible, just like always. However, now when encoding via the dumpers, `pvl` will default to writing out PDS3 Label Standard format PVL text, one of the strictest dialects, but other options are available. This behavior is different from the pre-1.0 version, which wrote out more generic PVL text.
- Removed the dependency on the `six` library that provided Python 2 compatibility.
- Removed the dependency on the `pytz` library that provided 'timezone' support, as that functionality is replaced with the Standard Library's `datetime` module.
- The private `pvl/_numbers.py` file was removed, as its capability is now accomplished with the Python Standard Library.
- The private `pvl/_datetimes.py` file was removed, as its capability is now accomplished with the Standard Library's `datetime` module.
- the private `pvl/_strings.py` file was removed, as its capabilities are now mostly replaced with the new grammar module and some functions in other new modules.
- Internally, the library is now working with string objects, not byte literals, so the `pvl/stream.py` module is no longer needed.
- Added an optional dependency on the 3rd party `dateutil` library, to parse more exotic date and time formats. If this library is not present, the `pvl` library will gracefully fall back to not parsing more exotic formats.
- Implmented a more formal approach to parsing PVL text: The properties of the PVL language are represented by a grammar object. A string is broken into tokens by the lexer function. Those tokens are parsed by a parser object, and when a token needs to be converted to a Python object, a decoder object does that job. When a Python object must be converted to PVL text, an encoder object does that job.
- Since the tests in `tests/test_decoder.py` and `tests/test_encoder.py` were really just exercising the loader and dumper functions, those tests were moved to `tests/test_pvl.py`, but all still work (with light modifications for the new defaults). Unit tests were added for most of the new classes and functions. All docstring tests now also pass doctest testing and are now included in the `make test` target.
- Functionality to correctly parse dash (-) continuation lines written by ISIS as detailed in #34 is implemented and tested.
- Functionality to use `pathlib.Path` objects for `pvl.load()` and `pvl.dump()` as requested in #20 and #31 is implemented and tested.
- Functionality to accept already-opened file objects that were opened in 'r' mode or 'rb' mode as alluded to in #6 is implemented and tested.
- The library now properly parses quoted strings that include backslashes as detailed in #33.
- Utility programs pvl_validate and pvl_translate were added.

- Documentation was updated and expanded.

10.20 0.3.0 (2017-06-28)

- Create methods to add items to the label
- Give user option to allow the parser to succeed in parsing broken labels

10.21 0.2.0 (2015-08-13)

- Drastically increase test coverage.
- Lots of bug fixes.
- Add Cube and PDS encoders.
- Cleanup README.
- Use pvl specification terminology.
- Added element access by index and slice.

10.22 0.1.1 (2015-06-01)

- Fixed issue with reading Pancam PDS Products.

10.23 0.1.0 (2015-05-30)

- First release on PyPI.

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W