PyBufrKit Documentation

Release 0.2.5

Yang Wang

May 11, 2021

Contents

PyBufrKit is a **pure** Python package to work with WMO BUFR (FM-94) messages. It can be used as both a command line tool or library to decode and encode BUFR messages. Here is a brief list of some of the features:

- Pure Python
- Handles both compressed and un-compressed messages
- Handles all practical operator descriptors, including data quality info, stats, bitmaps, etc.
- Option to construct hierarchial structure of a message, e.g. associate first order stats data to their owners.
- Convenient subsetting support for BUFR messages
- Comprehensive query support for BUFR messages
- Script support enables flexible extensions, e.g. filtering through large number of files.
- Tested with the same set of BUFR files used by ecCodes and BUFRDC.

More documentation at http://pybufrkit.readthedocs.io/

An online BUFR decoder powered by PyBufrKit, Serverless and AWS Lambda.

CHAPTER 1

Installation

PyBufrKit is compatible with Python 2.7, 3.5+, and PyPy. To install from PyPi:

```
pip install pybufrkit
```

Or from source:

```
python setup.py install
```

CHAPTER 2

Command Line Usage

The command line usage of the toolkit takes the following form:

```
pybufrkit [OPTIONS] command ...
```

where the `command` is one of following actions that can be performed by the tool:

- `decode` - Decode a BUFR file to outputs of various format, e.g. JSON
- `encode` - Encode a BUFR file from a JSON input
- `info` - Decode only the metadata sections (i.e. section 0, 1, 2, 3) of given BUFR files
- `split` - Split given BUFR files into one message per file
- `subset` - Subset the given BUFR file and save as new file
- `query` - Query metadata or data of given BUFR files
- `script` - Embed BUFR query expressions into normal Python script
- `lookup` - Look up information about the given list of comma separated BUFR descriptors
- `compile` - Compile the given comma separated BUFR descriptors

Here are a few examples using the tool from command line. For more details, please refer to the help option, e.g. `pybufrkit decode -h`. Also checkout the documentation.

```
# Decode a BUFR file and output in the default flat text format
pybufrkit decode BUFR_FILE

# Decode a file that is a concatenation of multiple BUFR messages,
# skipping any erroneous messages and continue on next one
pybufrkit decode -m --continue-on-error FILE

# Filter through a multi-message file and only decode messages
# that have data_category equals to 2. See below for details
# about usable filter expressions.
pybufrkit decode -m --filter '${%data_category} == 2' FILE
```

(continues on next page)

(continued from previous page)

```
# Decode a BUFR file and display it in a hierarchical structure
# corresponding to the BUFR Descriptors. In addition, the attribute
# descriptors are associated to their (bitmap) corresponding descriptors.
pybufrkit decode -a BUFR_FILE

# Decode a BUFR file and output in the flat JSON format
pybufrkit decode -j BUFR_FILE

# Encode from a flat JSON file to BUFR
pybufrkit encode -j JSON_FILE BUFR_FILE

# Decode a BUFR file, pipe it to the encoder to encode it back to BUFR
pybufrkit decode BUFR_FILE | pybufrkit encode -

# Decode only the metadata sections of a BUFR file
pybufrkit info BUFR_FILE

# Split a BUFR file into one message per file
pybufrkit split BUFR_FILE

# Subset from a given BUFR file
pybufrkit subset 0,3,6,9 BUFR_FILE

# Query values from the metadata sections (section 0, 1, 2, 3):
pybufrkit query %n_subsets BUFR_FILE

# Query all values for descriptor 001002 of the data section
pybufrkit query 001002 BUFR_FILE

# Query for those root level 001002 of the BUFR Template
pybufrkit query /001002 BUFR_FILE

# Query for 001002 that is a direct child of 301001
pybufrkit query /301001/001002 BUFR_FILE

# Query for all 001002 of the first subset
pybufrkit query '@[0] > 001002' BUFR_FILE

# Query for associated field of 021062
pybufrkit query 021062.A21062 BUFR_FILE

# Filtering through a number of BUFR files with Script support
# (find files that have multiple subsets):
pybufrkit script 'if ${%n_subsets} > 1: print(PBK_FILENAME)' DIRECTORY/*.bufr

# Lookup information for a Element Descriptor (along with its code table)
pybufrkit lookup -l 020003

# Compile a BUFR Template composed as a comma separated list of descriptors
pybufrkit compile 309052,205060
```

CHAPTER 3

Library Usage

The following code shows an example of basic library usage

```
# Decode a BUFR file
from pybufrkit.decoder import Decoder
decoder = Decoder()
with open(SOME_BUFR_FILE, 'rb') as ins:
    bufr_message = decoder.process(ins.read())

# Convert the BUFR message to JSON
from pybufrkit.renderer import FlatJsonRenderer
json_data = FlatJsonRenderer().render(bufr_message)

# Encode the JSON back to BUFR file
from pybufrkit.encoder import Encoder
encoder = Encoder()
bufr_message_new = encoder.process(json_data)
with open(BUFR_OUTPUT_FILE, 'wb') as outs:
    outs.write(bufr_message_new.serialized_bytes)

# Decode for multiple messages from a single file
from pybufrkit.decoder import generate_bufr_message
with open(SOME_FILE, 'rb') as ins:
    for bufr_message in generate_bufr_message(decoder, ins.read()):
        pass  # do something with the decoded message object

# Query the metadata
from pybufrkit.mdquery import MetadataExprParser, MetadataQuerent
n_subsets = MetadataQuerent(MetadataExprParser()).query(bufr_message, '%n_subsets')

# Query the data
from pybufrkit.dataquery import NodePathParser, DataQuerent
query_result = DataQuerent(NodePathParser()).query(bufr_message, '001002')

# Script
```

(continues on next page)

(continued from previous page)

```
from pybufrkit.script import ScriptRunner
# NOTE: must use the function version of print (Python 3), NOT the statement version
code = """print('Multiple' if ${%n_subsets} > 1 else 'Single')"""
runner = ScriptRunner(code)
runner.run(bufr_message)
```

For more help, please check the documentation site at http://pybufrkit.readthedocs.io/

CHAPTER 4

How It Works

4.1 BUFR Message Configuration files

The configurations describe how a BUFR message is composed of sections and how each section is organised. They are used to provide the overall structure definition of a BUFR message. The purpose of using configurations is to allow greater flexibility so that changes of BUFR Spec do NOT (to a certain extent) require program changes.

The builtin Configuration JSON files are located in the definitions directory inside the package. It can also be configured to load from an user provided directory. The naming convention of the files is as the follows:

```
sectionX[-Y].json
```

where X is the section index, Y is the edition number and optional.

Each section is configured with some metadata and a list of parameters. It takes the following general format:

```
{
  "index": 0,  # zero-based section index

  "description": "Indicator section",

  "default": true,  # use this config if an edition-specific one is not available

  "optional": false, # whether this section is optional

  "end_of_message": false, # whether this is the last section

  "parameters": [  # a list of parameter configs
    {
      "name": "start_signature",  # parameter name

      "nbits": 32,  # number of bits

      "type": "bytes",  # parameter type determines how the value can be processed␣
↪from the input bits
```

(continues on next page)

(continued from previous page)

```
      "expected": "BUFR",  # expected value for this parameter (will be validated if
↪not None)

      "as_property": false  # whether this parameter can be accessed from the parent
↪message object
    },
    ...
  ]
}
```

A few more notes about the configuration:

- Some section, e.g. Section 1, has edition-specific configuration, e.g. `section1-1.json`. However, most sections have a single configuration for all editions. The `default` field is used to indicate that the configuration is a catch-all for any editions that does not have its own specific config.
- Number of bits can be set to `0`, which means value of the corresponding parameter takes all the bits left for the section.
- **There are generally two categories of parameter types, simple and complex.**
  - Simple types are `uint` (unsigned integer), `int` (signed integer), `bool`, `bin` (binary bits) and `bytes` (string).
  - Complex types include `unexpanded_descriptors` and `template_data`. How they are processed is taken care of by a processor, e.g. Decoder. The configuration file does not concern how they are interpreted.
- The `expected` value will be validated against the actual value processed from the input. Currently, it is only used to ensure the start and stop signatures of a BUFR message.
- To allow loose coupling between sections, the parent message object can be configured to proxy some fields from a section. This is what the `as_property` field is for. For an example, the `edition` field from section 0 is needed for other sections to determine their structures. Therefore the `edition` field is proxyed by the parent message object so that it can be accessed by other sections without worrying about exactly which section provides this information.

4.2 Decoder and Encoder

These components process the input as prescribed by the configurations. Each sections are processed in order of the section index. The components also provide specialised methods to process parameters of complex types. The processing of `template_data` is where most of the program logic goes.

The `Decoder` and `Encoder` are sub-classes of the same abstract `Coder` class. They are implemented using the Template Method Pattern. The `Coder` base class provides bootstrapping and common functions needed by all of the sub-classes and leaves spaces for sub-classes to fill in the actual processing of the parameters.

For an example, the base class knows how to process an Element Descriptor. It prepares all necessary information about the descriptor, including its type, number of bits, units, scale, reference, etc. Depending on its type, the base class then invoke a method provided by the subclass to handle the actual processing, which can be either decoding or encoding.

4.3 BUFR Template Compilation

The main purpose of Template Compilation is performance. However since bit operations are the most time consuming part in the overall processing. The performance gain somewhat is limited. Depending on the total number of descriptors to be processed for a message, template compilation provides 10 - 30% performance boost. Larger number of descriptors, i.e. longer message, generally lead to less performance gain.

The implementation of `TemplateCompiler` is similar to the Decoder/Encoder. It is also a subclass of the abstract `Coder` class. It uses introspection to record method calls dispatched by the base class. The recorded calls can then be executed more efficiently because they bypass most of the BUFR template processing, such as checking descriptor type, expand sequence descriptors, etc.

4.4 Template Data Wiring

A BUFR Template is by nature hierarchical. For an example, a sequence descriptor has all of its child descriptors nested under it. When data associated to the template is decoded, they can also be organised in a hierarchical format. This is especially necessary when some operator descriptors, such as 204YYY (associated field) and bitmap related operators (222, 223, 224, 225, 232, 235, 236, 237), make some values as attributes to other values. The wiring process associates attributes to their owners so that their meanings are explicit.

4.4.1 Descriptors and Their IDs

In addition to the four canonical types of Descriptor defined by the BUFR Spec, this toolkit defines a few more Descriptors to help organise the decoded data. These new Descriptors are listed as follows:

- **Associated Descriptor** - This descriptor is created for the associated field that a Element Descriptor may have. Its 6-character ID is almost the same as the Element Descriptor except starting with a letter **A**. For an example, the ID of Associated Descriptor for Element Descriptor `015037` is `A15037`.
- **Skipped Local Descriptor** - This descriptor is used in place of any skipped local descriptors. This descriptor's ID begins with a letter **S** and the other 5 digits are the same of the descriptor that is skipped.
- **Marker Descriptors** - This is a group of Descriptors that are used in place of marker values such as substitution, first order stats etc. Their ID begins with **T**, **F**, **D**, and **R** for Substitution, First Order Stats, Difference Stats and Replacement/Retain, respectively. The other 5 digits are the same as the Element Descriptor they are associated via Bitmap.

4.5 Query BUFR Messages

Queries can be performed against either the metadata sections (section 0, 1, 2, 3) or the data section (the Template data). Though they are implemented separately in the backend, they share the same command line interface. This is made possible by requiring metadata query expression to always start with a percentage sign (%).

4.5.1 Query the Metadata Section

The following is the EBNF form of query expressions for metadata sections:

```
<query_expr> = '%'[<section_index>.]<parameter_name>
```

where the `parameter_name` are those defined in the configuration files, e.g. `n_subsets`, `edition`, etc.

The metadata query always return a scalar value. For parameters that are common across multiple sections, e.g. `section_length`, the first entry will be returned by default. For an example, the parameter `section_length` appears in Secton 1, 2, 3, and 4. By default, the entry of Section 1 is queried and its value is returned. To explicitly specify a Section, a `section_index` can be added in between the percentage sign and the `parameter_name`, e.g. `%2.section_length` returns the parameter value from Section 2 instead of 1.

4.5.2 Query the Template Data

The following is the EBNF form of query expressions for template data:

```
<query_expr> = [<subset_spec>] <path_spec>+
<subset_spec> = '@'<slice>
<path_spec> = <separator> <descriptor_id> [<slice>]
<separator> = '/' | '.' | '>'
```

- The `<slice>` takes the same syntax as how Python list can be sliced, e.g. `[1]`, `[-1]`, `[:]`, `[::10]`.
- The `<descriptor_id>` is the 6-letter/digit (leading zeros are required) descriptor ID, e.g. `001001`, `301001`, `A21062`.
- The `<separator>` can be omitted and defaults to > if a query string begins with a `<path_spec>`.
- Whitespaces are ignored.

The followings list a few examples of valid query expressions:

- `008042` - All instances of descriptor `008042` regardless of where it appears. This form is equivalent to `> 008042`.
- `@[0] > 008042` - Similar to the above query but only against the first subset.
- `/008042` - Only those that are root element of a BUFR Template
- `/008042[0]` - Similar to the above query but retrieve only the first instance. Note that the index does not account for the repetition of a descriptor in replication blocks, i.e. the descriptor will only be counted once.
- `303051/008042` - Only those that are direct children of `303051`
- `103000.031001` - The delayed replication factor value of replication `103000`. Note the separator between a delayed replication and its factor is a Dot.
- `021062.A21062` - The associated field of descriptor `021062`.

The query is performed against the wired hierarchical Template Data, which is *expanded*, *enhanced* and *populated*. These are explained as the follows:

- *Expanded* - The unexpanded descriptors are fully expanded. For an example, the sequence descriptor `301001` is expanded to contain two child descriptors, `001001` and `001002`. The hierarchical structure is also kept so that the child descriptor can be accurately specified using the Slash (`/`) separator.
- *Enhanced* - Associated fields, first order stats, bitmapped descriptors are wired as attributes to their owner descriptors. The attributes relationship can be queried using the Dot (`.`) separator.
- *Populated* - The Template is populated with actual data from the Data section. If a descriptor is not populated, for an example, a delayed replication block may have Zero replication, an empty list will be returned when any of its children is queried.

4.6 Script Support

Built upon the query feature, the script feature enables more flexible usage of the toolkit. The feature leverages full power of Python by embedding query expressions and injecting additional variables into normal Python code. For example, the following script filters for files that uses BUFR Template 309052:

```
if 309052 in ${%unexpanded_descriptors}: print(PBK_FILENAME)
```

Note that the query expressions are embedded into the code by enclosing them inside `${...}`. Also `PBK_FILENAME` is an extra variable injected by the toolkit to hold the name of current file being processed. Note you must use the function version of `print`. This is due to the use of `__future__` import in the code. But otherwise no Python 3 syntax is enforced.

You can also embed data queries like the follows:

```
print(${005001}, ${006001})
```

The above script prints latitude and longitude values from given BUFR files. One thing to note about data values is that they are by nature hierarchical. A file could contain multiple subsets, each subset could have replications. So the raw form of data values are nested list. However nested lists are rather difficult to work with and sometimes unnecessary. So it is possible to specify the nesting level of data values so they are easier to work with. By default, all values are turned into a simple list without any nesting. For an example, if each subset has one value for the given query, a list of N scalar values will be return with N equals to the number of subsets. This is referred as nesting level One as there is only one level of parenthesis for the returned value. All available nesting levels are:

- 0 - No parenthesis, only the first value will be returned as a scalar (all other values, if any, are simply dropped)
- 1 - One level of parenthesis (default). Values from all subsets are simply flattened into one simple list.
- 2 - Two level of parenthesis. Values from each subset are flatten into its own list, which is itself an element of the final return value.
- 4 - Fully hierarchical. No flatten at all. Each subset or replication have its own parenthesis grouping.

The above settings can be controlled via the command line option, `-n` or `--data-values-nest-level`. Alternatively it can also be specified with the script itself using following magic comment at the beginning:

```
#$ data_values_nest_level = 0
```

Note the magic comment line starts with `#$` and must appears before any other lines. The option passed from command line takes precedence over the option from the script itself.

4.7 Renderer

This component is responsible for rendering the processed BUFR message object in different formats, e.g. plain text, JSON.

CHAPTER 5

Show me the code

Check the code to see details.

5.1 API reference

5.1.1 pybufrkit

Work with WMO BUFR messages with Pure Python.

Note: APIs of version 0.2.0 are breaking changes from those of version 0.1.x and it is always recommended to upgrade to the latest version.

github https://github.com/ywangd/pybufrkit

docs http://pybufrkit.readthedocs.io/

author Yang Wang (ywangd@gmail.com)

5.1.2 pybufrkit.bufr

Classes for representing a BUFR message and its components.

class pybufrkit.bufr.**BufrMessage**(*filename=''*)

This class represents a single BUFR message that is comprised of different sections. Note this is different from BufrTemplateData which is only part of the overall message and dedicates to data associated to the Template.

Properties of this class are proxies to actual fields of its sections. They are set by the sections when they are processed. The proxy approach allows these properties to be referenced in a consistent way no matter where they actually come from. This makes sections loosely coupled, i.e. one section does not need to know about other sections, and free to change if needed.

add_section(*section*)
Add a section to the message

Parameters **section** (BufrSection) – The Bufr Section to add

build_template(*tables_root_dir*, *normalize=1*)
Build the BufrTemplate object using the list of unexpanded descriptors and corresponding table group.

Parameters

- **tables_root_dir** – The root directory to find BUFR tables
- **normalize** – Whether to use some default table group if the specific one is not available.

Returns A tuple of BufrTemplate and the associated TableGroup

wire()
Wire the flat list of descriptors and values to a full hierarchical structure. Also allocate all attributes to their corresponding descriptors.

class pybufrkit.bufr.**BufrSection**
This class represents a Section in a Bufr Message.

add_parameter(*parameter*)
Add a parameter to the section object.

Parameters **parameter** (SectionParameter) –

get_metadata(*k*)
Get value for metadata of the given name.

Parameters **k** (*str*) – Name of the metadata.

Returns Value of the metadata

get_parameter_offset(*parameter_name*)
Get the bit offset from the beginning of the section for parameter of the given name.

Returns The bit offset.

Return type int

set_metadata(*k*, *v*)
Set value to a metadata of the given key.

Parameters

- **k** (*str*) – Name of the metadata
- **v** (*object*) – Value of the metadata

class pybufrkit.bufr.**SectionConfigurer**(*definitions_dir=None*)
This class is responsible for loading the section config JSON files. It also initialise and configure a requested Section.

configure_section(*bufr_message*, *section_index*, *configuration_transformers=()*)
Initialise and Configure a section for the give section index and version.

Parameters

- **bufr_message** (BufrMessage) – The Bufr Message object to configure
- **section_index** (*int*) – (Zero-based) index of the section
- **configuration_transformers** (*collection*) – A collection of configuration transformation functions. These functions make it possible to use the same set of JSON files while still dynamically providing different coder behaviours.

Returns The configured section or `None` if not present

configure_section_with_values(*bufr_message*, *section_index*, *values*, *overrides=None*)
Initialise and Configure a section for the give section index and version and also populate the value of each section parameter with the given list of values. Used by the encoder.

Parameters

- **bufr_message** (`BufrMessage`) – The BUFR message object to configure
- **section_index** (*int*) – The zero-based section index
- **values** (*list*) – A list of values for the parameters.

Returns The configured section or `None` if the section is not present

static get_section_index_and_edition(*fname*)
Get Section Index and version from file name of a configuration file.

Parameters **fname** (*str*) – The base file name

Returns The index and edition numbers.

static ignore_value_expectation(*config*)
Remove any expectation value check.

Parameters **config** (*dict*) – The config JSON object loaded from a configuration file.

static info_configuration(*config*)
This is a configuration transformation function to make the decoder work only for the part of message before the template data.

Parameters **config** (*dict*) – The config JSON object loaded from a configuration file.

class pybufrkit.bufr.**SectionNamespace**(***kwds*)
A Section Namespace is an ordered dictionary that store the decoded parameters with their names as the keys.

class pybufrkit.bufr.**SectionParameter**(*name*, *nbits*, *data_type*, *expected*, *as_property*, *value=None*)
This class represents a Parameter of a Bufr Section.

5.1.3 pybufrkit.descriptors

The Descriptors should always be instantiated by Tables. Because the Tables provide caching and other wiring work. Do NOT instantiated the Descriptors directly!!

This module contains many Descriptor classes, covering not only the canonical descriptor types of the BUFR spec, but also Conceptual Descriptors that help the processing. For an example, an AssociatedDescriptor class is needed to represent associated values signified by operator descriptor 204YYY.

class pybufrkit.descriptors.**AssociatedDescriptor**(*id_*, *nbits*)
Associated field for element descriptor

Parameters **nbits** (*int*) – Number of bits used by this descriptor.

class pybufrkit.descriptors.**BufrTemplate**(*id_=999999*, *name=''*, *members=None*)
This class represents a BUFR Template. A Template is composed of one or more BUFR Descriptors. It is used in a BUFR message to describe the data section.

original_descriptor_ids
Get the list of descriptor IDs that can be used to instantiate the Template.

:rtype [int]

class pybufrkit.descriptors.**DelayedReplicationDescriptor**(*id_*, *members=None*, *factor=None*)

Delayed replication Descriptor 1XX000

class pybufrkit.descriptors.**Descriptor**(*id_*)

This class is the base class of all BUFR descriptors. It provides common machinery for Descriptors.

Parameters **id** (*int*) – The descriptor ID.

F

The F value of the descriptor.

X

The X value of the descriptor.

Y

THe Y value of the descriptor.

class pybufrkit.descriptors.**ElementDescriptor**(*id_*, *name*, *unit*, *scale*, *refval*, *nbits*, *crex_unit*, *crex_scale*, *crex_nchars*)

Element Descriptor 0XXYYY

Parameters

- **id** (*int*) – The descriptor ID
- **name** (*str*) – Name of the descriptor
- **unit** (*str*) – Units of the descriptor
- **scale** (*int*) – Scale factor of the descriptor value
- **refval** (*int*) – Reference value of the descriptor value
- **nbits** (*int*) – The number of bits used by the descriptor
- **crex_unit** (*str*) – Units of the descriptor for CREX spec
- **crex_scale** (*int*) – Scale factor of the descriptor value for CREX Spec
- **crex_nchars** (*int*) – Number of characters used by the descriptor for CREX Spec

class pybufrkit.descriptors.**FixedReplicationDescriptor**(*id_*, *members=None*)

Fixed replication Descriptor 1XXYYY

n_repeats

Number of times to perform the replication. This value is decoded directly from the descriptor ID.

class pybufrkit.descriptors.**MarkerDescriptor**(*id_*, *name*, *unit*, *scale*, *refval*, *nbits*, *crex_unit*, *crex_scale*, *crex_nchars*)

A marker descriptor is useful in the case when marker operator descriptors are used to signify a statistical value of an element descriptor. For an example, 224255 and 225255.

static from_element_descriptor(*ed*, *marker_id*, *scale=None*, *refval=None*, *nbits=None*)

Create from a given element descriptor with the option to override its scale, refval and nbits.

Parameters

- **ed** (ElementDescriptor) – The element descriptor
- **marker_id** (*int*) – The marker operator ID
- **scale** (*int*) – Overridden value for scale.
- **refval** (*int*) – Overridden value for reference.
- **nbits** (*int*) – Overridden value for number of bits.

Return type *MarkerDescriptor*

class pybufrkit.descriptors.**OperatorDescriptor**(*id_*)
Operator Descriptor 2XXYYY

class pybufrkit.descriptors.**ReplicationDescriptor**(*id_*, *members=None*)
The replication factor member stores only the replication factor descriptor NOT the actual value. So it is OK as it should be reusable for the same sequence descriptor. That is to say, when a Sequence Descriptor, e.g. 309052, is reused, the Replication Descriptor inside it should always have the same replication factor descriptor. Although these replication factor descriptor can have different values in different reuses of 309052, it does not matter as it does not store the actual values.

When the replication descriptor is reused as naked descriptor, i.e. not part of a Sequence Descriptor but directly under a Template, the associated replication factor descriptor could be different. But since the replication descriptor is NOT cached when used as naked. Every time a new Replication Descriptor is spawn thus there is no risk on the associated replication factor descriptor gets mixed up.

Parameters **members** (*[Descriptor]*) – The group of descriptors to be replicated

n_items
Number of descriptors to be replicated. This value is decoded from the ID of the descriptor.

n_members
Due to the hierarchical structure of the BUFR Template, The number of members is not always equal to number of items. For an example, the delayed replication factor counts towards number of items to be repeated for its outer replication (if nested). However, it will never be counted towards number of members. Other potential difference comes from Virtual descriptors, where virtual sequences and fixed replications are inserted/removed without fixing the enclosing replication descriptors. So in summary, the number of members is a more accurate count of the members to be replicated by the replication descriptor.

class pybufrkit.descriptors.**SequenceDescriptor**(*id_*, *name*, *members=None*)
Sequence Descriptor 3XXYYY

class pybufrkit.descriptors.**SkippedLocalDescriptor**(*id_*, *nbits*)
The skipped local descriptor is a placeholder for any descriptors followed by operator descriptor 206YYY.

class pybufrkit.descriptors.**UndefinedDescriptor**(*id_*)
Undefined Descriptors are only useful when loading BUFR tables that are NOT completely defined. For an example, an element descriptor is used by one of the sequence descriptor but the element descriptor itself is not defined in Table B. In this case, an Undefined descriptor is created in place of the actual element descriptor to allow tables to be loaded normally. As long as the Undefined descriptor is not used in the actual decoding (the Template of a BUFR message may not contain the descriptor at all), it is harmless to stay in the loaded Table Group.

Ideally this is not necessary if all tables are well defined. However, in practice, this is needed so some not-well-defined local tables can be used.

class pybufrkit.descriptors.**UndefinedElementDescriptor**(*id_*)

class pybufrkit.descriptors.**UndefinedSequenceDescriptor**(*id_*)

pybufrkit.descriptors.**flat_member_ids**(*descriptor*)
Return a flat list of expanded numeric IDs for the given descriptor. The list is generated by recursively flatten all its child members.

Parameters **descriptor** (Descriptor) – A BUFR descriptor

Returns [int]

5.1.4 pybufrkit.tables

The Table Cache makes sure tables of the same version only get loaded from disk once. When the get_table_group method is called, it returned a group of table either from the cache or loaded from disk if they are not available in the cache yet (and save them to the cache for future use).

- A Table Group contains a set of tables, e.g. A, B, C, D, that belong to the same table group key.
- A Table instance, e.g. B, D, maintains a cache of its descriptors so that only a single instance is created for an unique descriptor.
- The Pseudo Replication Descriptor table is created to make the API for all tables look alike.
- TableCache –creates–> TableGroup –lookup–> Descriptors/Template

Template are then processed by Coder in conjunction with a bit operator to create a BufrMessage object.

class pybufrkit.tables.**TableGroupKey**(*tables_root_dir*, *wmo_tables_sn*, *local_tables_sn*)

local_tables_sn
Alias for field number 2

tables_root_dir
Alias for field number 0

wmo_tables_sn
Alias for field number 1

5.1.5 pybufrkit.coder

class pybufrkit.coder.**AuditedList**
This class provides wrappers for some list methods, e.g. append, so that it is possible to execute additional code when the method is invoked. It is used mainly for debug purpose.

append(*p_object*)
L.append(object) – append object to end

class pybufrkit.coder.**BSRModifier**(*nbits_increment*, *scale_increment*, *refval_factor*)

nbits_increment
Alias for field number 0

refval_factor
Alias for field number 2

scale_increment
Alias for field number 1

class pybufrkit.coder.**Coder**(*definitions_dir=None*, *tables_root_dir=None*)
This class is an abstract superclass for Decoder and Encoder. By itself it cannot do anything. But it provides common operations for subclasses.

Parameters

- **definitions_dir** – Where to find the BPCL definition files.
- **tables_root_dir** – Where to find the BUFR table files.

define_bitmap(*state*, *reuse*)
Define a bit map.

Parameters

- **state** –
- **reuse** – Is this bitmap for reuse?

get_value_for_delayed_replication_factor(*state*)
Get value of the latest delayed replication factor. This is called when processing through the Template. But the actual implementation will be provided by sub-classes.

Parameters **state** –

Returns The value for the latest processed delayed replication factor

process(**args*, ***kwargs*)
Entry point of the class

process_associated_field(*state*, *bit_operator*, *descriptor*)

Parameters

- **bit_operator** –
- **descriptor** –

process_bitmap_definition(*state*, *bit_operator*, *descriptor*)
Process bitmap definition. This is basically done as a state machine for processing all the bits associated to the bitmap.

Parameters

- **bit_operator** –
- **descriptor** –

process_bitmapped_descriptor(*state*, *bit_operator*, *descriptor*)
A generic method for processing bitmapped descriptors. It is wrapped by providing different funcs to handle encoding and decoding for uncompressed and compressed data.

process_codeflag(*state*, *bit_operator*, *descriptor*, *nbits*)
Process a descriptor that has code/flag value. A code/flag value does not need to scale and refval.

Parameters

- **descriptor** – The BUFR descriptor
- **nbits** – Number of bits to process for the descriptor.

process_constant(*state*, *bit_operator*, *descriptor*, *value*)
Process a constant, with no bit operations, for the given descriptor.

Parameters

- **descriptor** – The BUFR descriptor.
- **value** – The constant value.

process_define_new_refval(*state*, *bit_operator*, *descriptor*)
Process defining a new reference value for the given descriptor.

Parameters

- **state** –
- **bit_operator** –
- **descriptor** –

process_delayed_replication_descriptor(*state*, *bit_operator*, *descriptor*)
Process the delayed replication factor descriptor.

Parameters

- **state** –
- **bit_operator** –

process_element_descriptor(*state*, *bit_operator*, *descriptor*)
Process an ElementDescriptor.

process_fixed_replication_descriptor(*state*, *bit_operator*, *descriptor*)
Process a fixed replication descriptor including all members belong to this replication structure.

Parameters

- **state** –
- **bit_operator** –

process_marker_operator_descriptor(*state*, *bit_operator*, *descriptor*)

Parameters

- **bit_operator** –
- **descriptor** –

process_members(*state*, *bit_operator*, *members*)
Process a list of descriptors that are members of a composite descriptor.

Parameters

- **state** – The state of the processing.
- **bit_operator** – The bit operator for read/write bits.
- **members** – A list of descriptors.

process_new_refval(*state*, *bit_operator*, *descriptor*, *nbits*)
Process the new reference value for the given descriptor.

Parameters

- **descriptor** – The BUFR descriptor.
- **nbits** – Number of bits to process.

process_numeric(*state*, *bit_operator*, *descriptor*, *nbits*, *scale_powered*, *refval*)
Process a descriptor that has numeric value.

Parameters

- **descriptor** – A BUFR descriptor that has numeric value
- **nbits** – Number of bits to process for the descriptor.
- **scale_powered** – 10 to the scale factor power, i.e. 10 ** scale
- **refval** – The reference value

process_numeric_of_new_refval(*state*, *bit_operator*, *descriptor*, *nbits*, *scale_powered*, *refval_factor*)
Process a descriptor that has numeric value with new reference value.

Parameters

- **descriptor** – The BUFR descriptor.
- **nbits** – Number of bits to process for the descriptor.
- **scale_powered** – 10 to the scale factor power, i.e. 10 ** scale
- **refval_factor** – The factor to be applied to the new refval.

process_operator_descriptor(*state*, *bit_operator*, *descriptor*)
Process Operator Descriptor.

Parameters

- **state** –
- **bit_operator** –

process_section(*bufr_message*, *bit_operator*, *section*)
Process the given section of a BUFR message

Parameters

- **bufr_message** – The BufrMessage object to process
- **bit_operator** – The bit operator (reader or writer)
- **section** –

process_skipped_local_descriptor(*state*, *bit_operator*, *descriptor*)
Skip number of bits defined for the local descriptor.

Parameters

- **state** –
- **bit_operator** –
- **descriptor** –

process_string(*state*, *bit_operator*, *descriptor*, *nbytes*)
Process a descriptor that has string value

Parameters

- **descriptor** – The BUFR descriptor
- **nbytes** – Number of BYTES to process for the descriptor.

process_template(*state*, *bit_operator*, *template*)
Process the top level BUFR Template

Parameters

- **state** – The state of the processing.
- **bit_operator** – The bit operator for read/write bits.
- **template** – The BUFR Template of the message.

class pybufrkit.coder.**CoderState**(*is_compressed*, *n_subsets*, *decoded_values_all_subsets=None*)
The state of Coder for keeping track of variables when a Coder is working. The use of a new state for each run makes it possible to use a single Coder to run multiple decoding/encoding tasks.

Parameters **decoded_values_all_subsets** – This is only for Encoder use.

add_bitmap_link()
Must be called before the descriptor is processed

build_bitmapped_descriptors(*bitmap*)
Build the bitmapped descriptors based on the given bitmap. Also build the back referenced descriptors if it is not already defined.

static minmax(*values*)
Give a list of values, find out the minimum and maximum, ignore any Nones.

switch_subset_context(*idx_subset*)
This function is only useful for uncompressed data.

5.1.6 pybufrkit.decoder

class pybufrkit.decoder.**Decoder**(*definitions_dir=None*, *tables_root_dir=None*, *compiled_template_cache_max=None*)
The decoder takes a bytes type string and decode it to a BUFR Message object.

define_bitmap(*state*, *reuse*)
For compressed data, bitmap and back referenced descriptors must be identical Otherwise it makes no sense in compressing different bitmapped descriptors into one slot.

Parameters

- **state** –
- **reuse** – Is this bitmap for reuse?

Returns The bitmap as a list of 0 and 1.

get_value_for_delayed_replication_factor(*state*)
Get value of the latest delayed replication factor. This is called when processing through the Template. But the actual implementation will be provided by sub-classes.

Parameters **state** –

Returns The value for the latest processed delayed replication factor

process(*s*, *file_path='<string>'*, *start_signature='BUFR'*, *info_only=False*, *ignore_value_expectation=False*, *wire_template_data=True*)
Decoding the given message string.

Parameters

- **s** – Message string that contains the BUFR Message
- **file_path** – The file where this string is read from.
- **start_signature** – Locate the starting position of the message string with the given signature.
- **info_only** – Only show information up to template data (exclusive)
- **ignore_value_expectation** – Do not validate the expected value
- **wire_template_data** – Whether to wire the template data to construct a fully hierarchical structure from the flat lists. Only takes effect when it is NOT info_only.

Returns A BufrMessage object that contains the decoded information.

process_codeflag(*state*, *bit_reader*, *descriptor*, *nbits*)
Process a descriptor that has code/flag value. A code/flag value does not need to scale and refval.

Parameters

- **descriptor** – The BUFR descriptor

- **nbits** – Number of bits to process for the descriptor.

process_constant(*state*, *bit_reader*, *descriptor*, *value*)
Process a constant, with no bit operations, for the given descriptor.

Parameters

- **descriptor** – The BUFR descriptor.
- **value** – The constant value.

process_new_refval(*state*, *bit_reader*, *descriptor*, *nbits*)
Process the new reference value for the given descriptor.

Parameters

- **descriptor** – The BUFR descriptor.
- **nbits** – Number of bits to process.

process_numeric(*state*, *bit_reader*, *descriptor*, *nbits*, *scale_powered*, *refval*)
Process a descriptor that has numeric value.

Parameters

- **descriptor** – A BUFR descriptor that has numeric value
- **nbits** – Number of bits to process for the descriptor.
- **scale_powered** – 10 to the scale factor power, i.e. 10 ** scale
- **refval** – The reference value

process_numeric_of_new_refval(*state*, *bit_reader*, *descriptor*, *nbits*, *scale_powered*, *refval_factor*)
Process a descriptor that has numeric value with new reference value.

Parameters

- **descriptor** – The BUFR descriptor.
- **nbits** – Number of bits to process for the descriptor.
- **scale_powered** – 10 to the scale factor power, i.e. 10 ** scale
- **refval_factor** – The factor to be applied to the new refval.

process_section(*bufr_message*, *bit_reader*, *section*)
Decode the given configured Section.

Parameters

- **bufr_message** – The BUFR message object.
- **section** – The BUFR section object.
- **bit_reader** –

Returns Number of bits decoded for this section.

process_string(*state*, *bit_reader*, *descriptor*, *nbytes*)
Process a descriptor that has string value

Parameters

- **descriptor** – The BUFR descriptor
- **nbytes** – Number of BYTES to process for the descriptor.

process_template_data(*bufr_message*, *bit_reader*)
Decode data described by the template.

Parameters

- **bufr_message** – The BUFR message object.
- **bit_reader** –

Returns TemplateData decoded from the bit stream.

process_unexpanded_descriptors(*bit_reader*, *section*)
Decode for the list of unexpanded descriptors.

Parameters

- **section** – The BUFR section object.
- **bit_reader** –

Returns The unexpanded descriptors as a list.

pybufrkit.decoder.**generate_bufr_message**(*decoder*, *s*, *info_only=False*, *continue_on_error=False*, *filter_expr=None*, **args*, ***kwargs*)
This is a generator function that processes the given string for one or more BufrMessage till it is exhausted.

Parameters

- **decoder** (Decoder) – Decoder to use
- **s** (*bytes*) – String to decode for messages

Returns BufrMessage object

5.1.7 pybufrkit.encoder

class pybufrkit.encoder.**Encoder**(*definitions_dir=None*, *tables_root_dir=None*, *ignore_declared_length=True*, *compiled_template_cache_max=None*, *master_table_number=None*, *master_table_version=None*)
The encoder takes a JSON object or string and encoded it to a BUFR message.

Parameters **ignore_declared_length** – If set, ignore the section_length declared in the input JSON message and always calculated it.

define_bitmap(*state*, *reuse*)
For compressed data, bitmap and back referenced descriptors must be identical Otherwise it makes no sense in compressing different bitmapped descriptors into one slot.

Parameters **reuse** – Is this bitmap for reuse?

get_value_for_delayed_replication_factor(*state*)
Get value of the latest delayed replication factor. This is called when processing through the Template. But the actual implementation will be provided by sub-classes.

Parameters **state** –

Returns The value for the latest processed delayed replication factor

process(*s*, *file_path='<string>'*, *wire_template_data=True*)
Entry point for the encoding process. The process encodes a JSON format message to BUFR message.

Parameters

- **s** – A JSON or its string serialized form
- **file_path** – The file path to the JSON file.
- **wire_template_data** – Whether to wire the template data to construct a fully hierarchical structure from the flat lists.

Returns A bitstring object of the encoded message.

process_codeflag(*state*, *bit_writer*, *descriptor*, *nbits*)
Process a descriptor that has code/flag value. A code/flag value does not need to scale and refval.

Parameters

- **descriptor** – The BUFR descriptor
- **nbits** – Number of bits to process for the descriptor.

process_codeflag_uncompressed(*state*, *bit_writer*, *descriptor*, *nbits*)
Decode a descriptor of code or flag value. A code or flag value does not need to scale and refval.

process_constant(*state*, *bit_writer*, *descriptor*, *value*)
Process a constant, with no bit operations, for the given descriptor.

Parameters

- **descriptor** – The BUFR descriptor.
- **value** – The constant value.

process_constant_compressed(*state*, *bit_writer*, *descriptor*, *value*)
This method is used for pop out value 0 for 222000, etc.

process_constant_uncompressed(*state*, *bit_writer*, *descriptor*, *value*)
This is in fact skip the value for encoding. Useful for operator descriptor 222000 etc.

process_new_refval(*state*, *bit_writer*, *descriptor*, *nbits*)
Process the new reference value for the given descriptor.

Parameters

- **descriptor** – The BUFR descriptor.
- **nbits** – Number of bits to process.

process_numeric(*state*, *bit_writer*, *descriptor*, *nbits*, *scale_powered*, *refval*)
Process a descriptor that has numeric value.

Parameters

- **descriptor** – A BUFR descriptor that has numeric value
- **nbits** – Number of bits to process for the descriptor.
- **scale_powered** – 10 to the scale factor power, i.e. 10 ** scale
- **refval** – The reference value

process_numeric_of_new_refval(*state*, *bit_writer*, *descriptor*, *nbits*, *scale_powered*, *refval_factor*)
Encode a descriptor of numeric value that has a new reference value set by 203 YYY. This new reference value must be retrieved at runtime as it is defined in the data section.

Parameters **refval_factor** (*int*) – The refval factor set as part of 207 YYY

Returns

process_section(*bufr_message*, *bit_writer*, *section*)
Process the given section of a BUFR message

Parameters

- **bufr_message** – The BufrMessage object to process
- **bit_operator** – The bit operator (reader or writer)
- **section** –

process_string(*state*, *bit_writer*, *descriptor*, *nbytes*)
Process a descriptor that has string value

Parameters

- **descriptor** – The BUFR descriptor
- **nbytes** – Number of BYTES to process for the descriptor.

process_string_uncompressed(*state*, *bit_writer*, *descriptor*, *nbytes*)
Decode a string value of the given number of bytes

Parameters

- **descriptor** –
- **nbytes** – Number of bytes to read for the string.

process_template_data(*bufr_message*, *bit_writer*, *section_parameter*)

Parameters **bit_writer** –

Returns

process_unexpanded_descriptors(*bit_writer*, *section_parameter*)
Encode the list of unexpanded descriptors. :type bit_writer: bitops.BitWriter :type section_parameter: bufr.SectionParameter

5.1.8 pybufrkit.templatecompiler

pybufrkit.templatecompiler.**loads_compiled_template**(*s*)
Load a compiled template object from its JSON string representation.

Parameters **s** – A JSON string represents the compiled template.

Returns The compiled template

class pybufrkit.templatecompiler.**TemplateCompiler**
The compiler for the BUFR Template. This class does its job by recording calls from the generic Coder.

define_bitmap(*state*, *reuse*)
Define a bit map.

Parameters

- **state** –
- **reuse** – Is this bitmap for reuse?

get_value_for_delayed_replication_factor(*state*)
Get value of the latest delayed replication factor. This is called when processing through the Template. But the actual implementation will be provided by sub-classes.

Parameters **state** –

Returns The value for the latest processed delayed replication factor

process (*template*, *table_group*)
Entry point of the Compiler.

Parameters

- **template** (descriptors.BufrTemplate) – The BUFR template to compile
- **table_group** (*tables.TableGroup*) – The Table Group used to instantiate the Template.

Returns CompiledTemplate

process_bitmap_definition (*state*, *bit_operator*, *descriptor*)
Process bitmap definition. This is basically done as a state machine for processing all the bits associated to the bitmap.

Parameters

- **bit_operator** –
- **descriptor** –

process_bitmapped_descriptor (*state*, *bit_operator*, *descriptor*)
A generic method for processing bitmapped descriptors. It is wrapped by providing different funcs to handle encoding and decoding for uncompressed and compressed data.

process_codeflag (*state*, *bit_operator*, *descriptor*, *nbits*)
Process a descriptor that has code/flag value. A code/flag value does not need to scale and refval.

Parameters

- **descriptor** – The BUFR descriptor
- **nbits** – Number of bits to process for the descriptor.

process_constant (*state*, *bit_operator*, *descriptor*, *value*)
Process a constant, with no bit operations, for the given descriptor.

Parameters

- **descriptor** – The BUFR descriptor.
- **value** – The constant value.

process_delayed_replication_descriptor (*state*, *bit_operator*, *descriptor*)
Process the delayed replication factor descriptor.

Parameters

- **state** –
- **bit_operator** –

process_fixed_replication_descriptor (*state*, *bit_operator*, *descriptor*)
Process a fixed replication descriptor including all members belong to this replication structure.

Parameters

- **state** –
- **bit_operator** –

process_new_refval (*state*, *bit_operator*, *descriptor*, *nbits*)
Process the new reference value for the given descriptor.

Parameters

- **descriptor** – The BUFR descriptor.
- **nbits** – Number of bits to process.

process_numeric(*state*, *bit_operator*, *descriptor*, *nbits*, *scale_powered*, *refval*)
Process a descriptor that has numeric value.

Parameters

- **descriptor** – A BUFR descriptor that has numeric value
- **nbits** – Number of bits to process for the descriptor.
- **scale_powered** – 10 to the scale factor power, i.e. 10 ** scale
- **refval** – The reference value

process_numeric_of_new_refval(*state*, *bit_operator*, *descriptor*, *nbits*, *scale_powered*, *refval_factor*)
Process a descriptor that has numeric value with new reference value.

Parameters

- **descriptor** – The BUFR descriptor.
- **nbits** – Number of bits to process for the descriptor.
- **scale_powered** – 10 to the scale factor power, i.e. 10 ** scale
- **refval_factor** – The factor to be applied to the new refval.

process_section(*bufr_message*, *bit_operator*, *section*)
Process the given section of a BUFR message

Parameters

- **bufr_message** – The BufrMessage object to process
- **bit_operator** – The bit operator (reader or writer)
- **section** –

process_string(*state*, *bit_operator*, *descriptor*, *nbytes*)
Process a descriptor that has string value

Parameters

- **descriptor** – The BUFR descriptor
- **nbytes** – Number of BYTES to process for the descriptor.

class pybufrkit.templatecompiler.**CompiledTemplateManager**(*cache_max*)
A management class for compiled templates that handles caching and lookup.

Parameters **cache_max** – The maximum number of compiled templates to cache.

get_or_compile(*template*, *table_group*)

Parameters

- **template** (descriptors.BufrTemplate) – The BUFR template to compile
- **table_group** (*tables.TableGroup*) – The Table Group used to instantiate the Template.

Returns

pybufrkit.templatecompiler.**process_compiled_template**(*coder*, *state*, *bit_operator*, *compiled_template*)

This function runs the compiled code from the TemplateCompiler

Parameters

- **coder** (Coder) –
- **state** (*VmState*) –
- **bit_operator** –
- **compiled_template** (*Block*) –

5.1.9 pybufrkit.templatedata

The TemplateData object is dedicated to the data decoded for the template of a BUFR message, while Bufr object is for the entire BUFR message. The object provides a fully hierarchical view of the data with attributes properly allocated to their corresponding values.

class pybufrkit.templatedata.**AssociatedFieldNode**(*descriptor*, *index*)

class pybufrkit.templatedata.**DataNode**(*descriptor*)

A node is composed of a descriptor and its value (if exists) and any possible child or attribute nodes.

class pybufrkit.templatedata.**DelayedReplicationNode**(*descriptor*)

class pybufrkit.templatedata.**DifferenceStatsNode**(*descriptor*, *index*)

class pybufrkit.templatedata.**FirstOrderStatsNode**(*descriptor*, *index*)

class pybufrkit.templatedata.**FixedReplicationNode**(*descriptor*)

class pybufrkit.templatedata.**NoValueDataNode**(*descriptor*)

A no value node is for any descriptors that cannot have a value, e.g. replication descriptors, sequence descriptors and some operator descriptors, e.g. 201YYY.

class pybufrkit.templatedata.**QualityInfoNode**(*descriptor*, *index*)

class pybufrkit.templatedata.**ReplacementNode**(*descriptor*, *index*)

class pybufrkit.templatedata.**SequenceNode**(*descriptor*)

class pybufrkit.templatedata.**SubstitutionNode**(*descriptor*, *index*)

class pybufrkit.templatedata.**TemplateData**(*template*, *is_compressed*, *decoded_descriptors_all_subsets*, *decoded_values_all_subsets*, *bitmap_links_all_subsets*)

This class is dedicated to the data section of a BUFR message and produces a fully hierarchical structure for the otherwise flat list of decoded descriptors and values. Attributes like associated fields and statistical values are properly allocated to their corresponding referred elements.

wire()

From the flat list of descriptors and values, construct a fully hierarchical structure of data including sequence descriptors and correctly set bitmapped values to their corresponding node as attributes.

wire_delayed_replication_descriptor(*descriptor*)

Parameters **descriptor** (DelayedReplicationDescriptor) –

wire_fixed_replication_descriptor(*descriptor*)

Parameters **descriptor** (FixedReplicationDescriptor) –

wire_operator_descriptor(*descriptor*)

Parameters **descriptor** (OperatorDescriptor) –

Returns

class pybufrkit.templatedata.**ValueDataNode**(*descriptor*, *index*)

A value node is for any descriptors that can have a value attached to it. This includes all Element descriptor, Associated descriptor, Skipped local descriptor, some operator descriptors, e.g. 205YYY, 223255, etc.

Parameters **index** (*int*) – The index to the descriptors and values array for getting the descriptor and its associated value.

5.1.10 pybufrkit.dataquery

class pybufrkit.dataquery.**NodePathParser**(*bare_id_matches_all=True*)

This class provides a parser for parsing path query string.

Parameters **bare_id_matches_all** (*bool*) – By default, a path component with bare ID, i.e. with no slicing part, means match all occurrences, i.e. [::]. If set to False, it only matches the first occurence.

class pybufrkit.dataquery.**DataQuerent**(*path_parser*)

This class provides interface to query the BUFR Data section.

create_values_from_nodes(*nodes*, *decoded_values*)

Process through the nested matching node list and create an values list of identical structure. This method is recursive.

Parameters

- **nodes** – A nested list of matching nodes.
- **decoded_values** –

Returns A nested values list corresponding to the given nodes.

descend_and_proceed(*nodes*, *path_components*)

Processing through the given list of nodes, for any nodes that are not a direct match of the path component, descent to its sub-nodes for further matching. Once a match is found, it then proceed through the path component till all the component is matched or zero match is encountered.

Parameters

- **nodes** – A list of nodes to descend into its sub-nodes
- **path_components** – The path components used for matching.

filter_for_attribute_sub_nodes(*node*, *path_components*)

This method is a specific version of the filter_for_sub_nodes method. It first filters through the attribute nodes of the given node and then goes depth first till all the path components are matched or zero match is found.

filter_for_child_sub_nodes(*node*, *path_components*)

This method is a specific version of the filter_for_sub_nodes method. It first filters through the child nodes of the given node and then goes depth first till all the path components are matched or zero match is found.

filter_for_descendant_sub_nodes(*node*, *path_components*)

This method is a specific version of the filter_for_sub_nodes method. It filter through all descendant nodes in a depth first fashion of the given node. A descendant node could be either a child, attribute or factor node all the way to the leaf node. It then process through path components till every component is matched or zero match is encountered.

filter_for_nodes(*nodes*, *path_component*)
Filter the given list of nodes using the path component. Note this method is different from the filter_for_sub_nodes method in that it filters the given nodes themselves, NOT their sub-nodes. The return value will be a selection of the given nodes.

Parameters

- **nodes** – A list of nodes to be filtered
- **path_component** – The path component used for the filtering.

Returns A list of nodes that qualified by the path component.

filter_for_sub_nodes(*node*, *path_components*)
For the given node, filter through its sub-nodes, which could be child, attribute, factor or descendant nodes depending on the separator value of the first member of path components. Note that the filtering will be performed in a depth first fashion, i.e. the filtering is continued with the direct sub-nodes down to the leaves of the node tree or the end of path components, whichever encounters first.

Parameters

- **node** – The node for which the sub-nodes will be filtered
- **path_components** – A list of path components used to filter the nodes.

Returns A list of qualified nodes matching through the entire path components.

node_matches(*node*, *path_component*)
Check whether the given node is qualified with the path component. If the path component's separator is descendant, any sub-nodes containing node is qualified.

Parameters

- **node** –
- **path_component** –

Returns True or False

proceed_next_path_component(*nodes*, *path_components*)
Proceed further down the path components.

Parameters

- **nodes** –
- **path_components** –

Returns

query(*bufr_message*, *path_expr*)
Entry method of the class. Query the data section of the given BUFR message with the query string.

Parameters

- **bufr_message** – A BufrMessage object with wired nodes
- **path_expr** – A query string for data.

Returns A QueryResult object

class pybufrkit.dataquery.**QueryResult**(*path_expr=''*)
This class represents the query result.

5.1.11 pybufrkit.mdquery

class pybufrkit.mdquery.**MetadataQuerent**(*metadata_expr_parser*)

Parameters **metadata_expr_parser** (*MetadataExprParser*) – Parser for metadata expression

5.1.12 pybufrkit.query

class pybufrkit.query.**BufrMessageQuerent**

This is a convenient class for querents of metadata and data sections. It provides an uniform interface for querying the BufrMessage object.

5.1.13 pybufrkit.script

pybufrkit.script.**process_embedded_query_expr**(*input_string*)

This function scans through the given script and identify any path/metadata expressions. For each expression found, an unique python variable name will be generated. The expression is then substituted by the variable name.

Parameters **input_string** (*str*) – The input script

Returns A 2-element tuple of the substituted string and a dict of substitutions

Return type (str, dict)

class pybufrkit.script.**ScriptRunner**(*input_string*, *data_values_nest_level=None*, *mode='exec'*)

This class is responsible for running the given script against BufrMessage object.

code_string
The processed/substituted source code.

code_object
The compiled code object from the code string.

pragma
Extra processing directives

metadata_only
Whether the script requires only metadata part of the BUFR message to work.

querent
The BufrMessageQuerent object for performing the values query.

5.1.14 pybufrkit.renderer

class pybufrkit.renderer.**FlatJsonRenderer**

This renderer converts the given object to a JSON string by flatten its internal structure.

class pybufrkit.renderer.**FlatTextRenderer**

This renderer converts the given object by flatten all its sub-structures.

class pybufrkit.renderer.**NestedJsonRenderer**

The counterpart to NestedTextRenderer but with JSON as output

class pybufrkit.renderer.**NestedTextRenderer**

This renderer converts the given object to a text string by honoring all its nested sub-structures.

class pybufrkit.renderer.**Renderer**
This class is the abstract base Renderer. A renderer provides the contract to take in an object and convert it into a string representation.

render(*obj*)
Render the given object as string.

Parameters **obj** (*object*) – The object to render

Returns A string representation of the given object.

Return type str

5.1.15 pybufrkit.bitops

class pybufrkit.bitops.**BitStringBitReader**(*s*)
A BitReader implementation using the bitstring module.

Parameters **bitstring.BitStream** (*bit_stream*) – Bit stream created from the input string

get_pos()
Retrieve the bit position for next read

read_bin(*nbits*)
Read number of bits as bytes representation of binary number

read_bool()
Read one bit for value of boolean

read_bytes(*nbytes*)
Read number of bytes for value of bytes type

read_int(*nbits*)
Read number of bits as integer

read_uint(*nbits*)
Read number of bites for value of unsigned integer

class pybufrkit.bitops.**BitStringBitWriter**
A BitWriter implementation using the bitstring module.

get_pos()
Retrieve the bit position for next write

set_uint(*value*, *nbits*, *bitpos*)
Set an unsigned integer value of given number of bits at the bit position and replace the old value.

skip(*nbits*)
Skip ahead for the given number of nbits

to_bytes()
dump all content to bytes type

write_bin(*value*)
Write a binary number represented by the given value. The length is determined by the value.

write_bool(*value*)
Write one bit for value of boolean type

write_bytes(*value*, *nbytes=None*)
Write given number of bits value of bytes type. If nbytes is none, use the length of the given bytes value

write_int(*value*, *nbits*)
Write given number of bits for value of signed integer

write_uint(*value*, *nbits*)
Write given number of bits value of unsigned integer type

pybufrkit.bitops.**get_bit_reader**(*s*)
Initialise and return a BitReader the given string. This function is intended to shield the actual implementation of BitReader away from the caller.

Parameters **s** – The byte string to read from.

Returns BitReader

pybufrkit.bitops.**get_bit_writer**()
Initialise and return a BitWriter.

Returns BitWriter

5.1.16 pybufrkit.commands

This file gathers all the functions that support the command line usages.

pybufrkit.commands.**command_decode**(*ns*)
Command to decode given files from command line.

pybufrkit.commands.**command_info**(*ns*)
Command to show metadata information of given files from command line.

pybufrkit.commands.**command_encode**(*ns*)
Command to encode given JSON file from command line into BUFR file.

pybufrkit.commands.**command_lookup**(*ns*)
Command to lookup the given descriptors from command line

pybufrkit.commands.**command_compile**(*ns*)
Command to compile the given descriptors.

pybufrkit.commands.**command_subset**(*ns*)
Command to subset and save the given BUFR file.

pybufrkit.commands.**command_query**(*ns*)
Command to query given BUFR files.

pybufrkit.commands.**command_script**(*ns*)
Command to execute script against given BUFR files.

pybufrkit.commands.**command_split**(*ns*)
Command to split given files from command line into one file per BufrMessage.

5.1.17 pybufrkit.utils

class pybufrkit.utils.**EntityEncoder**(*skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *encoding='utf-8'*, *default=None*)

default(*o*)
Implement this method in a subclass such that it returns a serializable object for `o`, or calls the base implementation (to raise a `TypeError`).

For example, to support arbitrary iterators, you could implement default like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return JSONEncoder.default(self, o)
```

pybufrkit.utils.**fixed_width_repr_of_int**(*value*, *width*, *pad_left=True*)
Format the given integer and ensure the result string is of the given width. The string will be padded space on the left if the number is small or replaced as a string of asterisks if the number is too big.

Parameters

- **value** (*int*) – An integer number to format
- **width** (*int*) – The result string must have the exact width

Returns A string representation of the given integer.

pybufrkit.utils.**flat_text_to_flat_json**(*flat_text*)
Convert the flat Text output to the flat JSON output format

Parameters **flat_text** (*str*) – The flat text output

pybufrkit.utils.**flatten_list**(*values*)
Flatten a list so everything is in a list without nesting . :param values: :return:

pybufrkit.utils.**generate_quiet**(*iterable*, *next_val*)
Iterate, returning if the generator function raises StopIteration.

https://www.python.org/dev/peps/pep-0479/

pybufrkit.utils.**nested_json_to_flat_json**(*nested_json_data*)
Converted the nested JSON output to the flat JSON output. This is useful as Encoder only works with flat JSON.

Parameters **nested_json_data** – The nested JSON object

Returns Flat JSON object.

pybufrkit.utils.**nested_text_to_flat_json**(*nested_text*)
Convert string in nested text format to a flat JSON object. :param str nested_text: The nested text output :return: A flat JSON object

pybufrkit.utils.**section_text_to_flat_json**(*lines*, *idxline*, *func_subsets_text_to_flat_json*)
Convert a section from text output to a section of flat JSON.

pybufrkit.utils.**subsets_flat_text_to_flat_json**(*lines*, *idxline*)
Convert all subsets data from flat text output to all subsets data of flat JSON.

pybufrkit.utils.**subsets_nested_text_to_flat_json**(*lines*, *idxline*)
Convert all subsets data from nested text format to flat JSON format.

pybufrkit.utils.**template_data_nested_json_to_flat_json**(*template_data_value*)
Helper function to convert nested JSON of template data to flat JSON.

5.1.18 pybufrkit.constants

Various constants used in the module.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

E

F

G

I

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y