## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  Reading</td>
<td>3</td>
</tr>
<tr>
<td>2  Writing</td>
<td>5</td>
</tr>
<tr>
<td>3  Updating</td>
<td>7</td>
</tr>
<tr>
<td>4  JSON and XML</td>
<td>9</td>
</tr>
<tr>
<td>5  API Docs</td>
<td>11</td>
</tr>
<tr>
<td>5.1 Reader</td>
<td>11</td>
</tr>
<tr>
<td>5.2 Record</td>
<td>13</td>
</tr>
<tr>
<td>5.3 Writer</td>
<td>16</td>
</tr>
<tr>
<td>5.4 Exceptions</td>
<td>20</td>
</tr>
<tr>
<td>5.5 MarcXML</td>
<td>21</td>
</tr>
<tr>
<td>5.6 Constants</td>
<td>22</td>
</tr>
<tr>
<td>5.7 MARC-8</td>
<td>22</td>
</tr>
<tr>
<td>5.8 MARC-8 mapping</td>
<td>23</td>
</tr>
<tr>
<td>6  Indices and tables</td>
<td>25</td>
</tr>
<tr>
<td>Python Module Index</td>
<td>27</td>
</tr>
<tr>
<td>Index</td>
<td>29</td>
</tr>
</tbody>
</table>
Release v5.1.0

Pymarc is a Python 3 library for working with bibliographic data encoded in MARC21.

Starting with version 5.0.0 it requires python 3.7 and up. It provides an API for reading, writing and modifying MARC records. It was mostly designed to be an emergency eject seat, for getting your data assets out of MARC and into some kind of saner representation. However over the years it has been used to create and modify MARC records, since despite repeated calls for it to die as a format, MARC seems to be living quite happily as a zombie.

Below are some common examples of how you might want to use pymarc. If you run across an example that you think should be here please send a pull request.
Most often you will have some MARC data and will want to extract data from it. Here’s an example of reading a batch of records and printing out the title. If you are curious this example uses the batch file available here in pymarc repository:

```python
from pymarc import MARCReader

with open('test/marc.dat', 'rb') as fh:
    reader = MARCReader(fh)
    for record in reader:
        print(record.title)
```

The pragmatic programmer: from journeyman to master / Programming Python / Learning Python / Python cookbook / Python programming for the absolute beginner / Web programming: techniques for integrating Python, Linux, Apache, and MySQL / Python programming on Win32 / Python programming: an introduction to computer science / Python Web programming / Core python programming / Python and Tkinter programming / Game programming with Python, Lua, and Ruby / Python programming patterns / Python programming with the Java class libraries: a tutorial for building Web and Enterprise applications / Learn to program using Python: a tutorial for hobbyists, self-starters, and all who want to learn the art of computer programming / Programming with Python / BSD Sockets programming from a multi-language perspective / Design patterns: elements of reusable object-oriented software / Introduction to algorithms / ANSI Common Lisp /

Sometimes MARC data contains an errors of some kind. In this case reader returns None instead of record object and
two reader’s properties `current_exception` and `current_chunk` can help the user to take a corrective action and continue or stop the reading:

```python
from pymarc import MARCReader
from pymarc import exceptions as exc

with open('test/marc.dat', 'rb') as fh:
    reader = MARCReader(fh)
    for record in reader:
        if record:
            # consume the record:
            print(record.title)
        elif isinstance(reader.current_exception, exc.FatalReaderError):
            # data file format error
            print(reader.current_exception)
            print(reader.current_chunk)
        else:
            # fix the record data, skip or stop reading:
            print(reader.current_exception)
            print(reader.current_chunk)
            # break/continue/raise
```

`FatalReaderError` happens when reader can’t determine record’s boundaries in the data stream. To avoid data misinterpretation it stops. In case of other errors (wrong encoding etc.) reader continues to the next record.

A `pymarc.Record` object has a few handy properties like `title` for getting at bits of a bibliographic record, others include: `author`, `isbn`, `subjects`, `location`, `notes`, `physicaldescription`, `publisher`, `pubyear`. But really, to work with MARC data you need to understand the numeric field tags and subfield codes that are used to designate various bits of information. There is a lot more hiding in a MARC record than these properties provide access to. For example the title property extracts the information from the 245 field, subfields a and b. You can access 245a like so:

```python
print(record['245']['a'])
```

Some fields like `subjects` can repeat. In cases like that you will want to use `get_fields` to get all of them as `pymarc.Field` objects, which you can then interact with further:

```python
for f in record.get_fields('650'):
    print(f)
```

If you are new to MARC fields, “Understanding MARC” (http://www.loc.gov/marc/umb/) is a pretty good primer, and the “MARC 21 Formats” (http://www.loc.gov/marc/marcdocz.html) page at the Library of Congress is a good reference once you understand the basics.

**Note:** New in v5.0.0, `Subfield` is used to create subfields. Prior to v5, subfields were constructed as a list of strings, e.g., `[code, value, code, value]`. This has been changed to organize the subfields into a list of tuples, e.g., `[(`code, value`)]`. The `Subfield` is implemented as a `NamedTuple` so that the tuples can be constructed as `Subfield(code=code, value=value)`. See the code below for an example of how this is used.

The old style of creating subfields is no longer supported. Passing a list of strings to the `subfields` parameter for the `Field` constructor will raise a `ValueError`.

For convenience, a class method is provided to convert the legacy list of strings into a list of `Subfield`’s. An example of how to do this is given below.
Here’s an example of creating a record and writing it out to a file.

```python
from pymarc import Record, Field, Subfield

record = Record()
record.add_field(
    Field(
        tag = '245',
        indicators = ['0','1'],
        subfields = [
            Subfield(code='a', value='The pragmatic programmer : '),
            Subfield(code='b', value='from journeyman to master /'),
            Subfield(code='c', value='Andrew Hunt, David Thomas.')
        ])
)

with open('file.dat', 'wb') as out:
    out.write(record.as_marc())
```
Updating works the same way, you read it in, modify it, and then write it out again:

```python
from pymarc import MARCReader

with open('test/marc.dat', 'rb') as fh:
    reader = MARCReader(fh)
    record = next(reader)
    record['245']['a'] = 'The Zombie Programmer : '

with open('file.dat', 'wb') as out:
    out.write(record.as_marc())
```
If you find yourself using MARC data a fair bit, and distributing it, you may make other developers a bit happier by using the JSON or XML serializations. pymarc has support for both. The main benefit here is that the UTF8 character encoding is used, rather than the frustratingly archaic MARC8 encoding. Also they will be able to use JSON and XML tools to get at the data they want instead of some crazy MARC processing library like, ahem, pymarc.
CHAPTER 5

API Docs

5.1 Reader

Pymarc Reader.

```python
class pymarc.reader.JSONReader(marc_target: Union[bytes, str], encoding: str = 'utf-8', stream: bool = False)
    Bases: pymarc.reader.Reader
    JSON Reader.

class pymarc.reader.MARCReader(marc_target: Union[BinaryIO, bytes], to_unicode: bool = True, force_utf8: bool = False, hide_utf8_warnings: bool = False, utf8_handling: str = 'strict', file_encoding: str = 'iso8859-1', permissive: bool = False)
    Bases: pymarc.reader.Reader
    An iterator class for reading a file of MARC21 records.
    Simple usage:
```

```python
from pymarc import MARCReader

## pass in a file object
reader = MARCReader(open('file.dat', 'rb'))
for record in reader:
    ...

## pass in marc in transmission format
reader = MARCReader(rawmarc)
for record in reader:
    ...
```

If you would like to have your Record object contain unicode strings use the to_unicode parameter:
reader = MARCReader(open('file.dat', 'rb'), to_unicode=True)

This will decode from MARC-8 or utf-8 depending on the value in the MARC leader at position 9. Upon serialization of the Record object to MARC21, the resulting output will be utf-8 encoded and the value in the MARC leader at position 9 will be set appropriately to indicate the change of character encoding.

If you find yourself in the unfortunate position of having data that is utf-8 encoded without the leader set appropriately you can use the force_utf8 parameter:

```python
reader = MARCReader(open('file.dat', 'rb'), to_unicode=True, force_utf8=True)
```

If you find yourself in the unfortunate position of having data that is mostly utf-8 encoded but with a few non-utf-8 characters, you can also use the utf8_handling parameter, which takes the same values (‘strict’, ‘replace’, and ‘ignore’) as the Python Unicode codecs (see http://docs.python.org/library/codecs.html for more info).

Although, it’s not legal in MARC-21 to use anything but MARC-8 or UTF-8, but if you have a file in incorrect encode and you know what it is, you can try to use your encode in parameter “file_encoding”.

MARCReader parses data in a permissive way and gives the user full control on what to do in case wrong record is encountered. Whenever any error is found reader returns None instead of regular record object. The exception information and corresponding data are available through reader.current_exception and reader.current_chunk properties:

```python
reader = MARCReader(open('file.dat', 'rb'))
for record in reader:
    if record is None:
        print("Current chunk: ",
              record.current_chunk,
              " was ignored because the following exception raised: ",
              record.current_exception)
    else:
        # do something with record
```

close() → None
Close the handle.

current_chunk
Current chunk.

current_exception
Current exception.

class pymarc.reader.Reader
Bases: object
A base class for all iterating readers in the pymarc package.

pymarc.reader.map_records(f: Callable, *files) → None
Applies a given function to each record in a batch.
You can pass in multiple batches.

```python
def print_title(r):
    print(r['245'])
map_records(print_title, file('marc.dat'))
```
## 5.2 Record

Pymarc Record.

```python
class pymarc.record.Record(data: str = ",
    to_unicode: bool = True, force_utf8: bool = False,
    hide UTF8_warnings: bool = False, utf8_handling: str = 'strict',
    leader: str = ' ', file_encoding: str = 'iso8859-1')
```

Bases: object

A class for representing a MARC record.

Each Record object is made up of multiple Field objects. You’ll probably want to look at the docs for Field to see how to fully use a Record object.

Basic usage:

```python
field = Field(
    tag = '245',
    indicators = ['0', '1'],
    subfields = [
        Subfield(code='a', value='The pragmatic programmer : '),
        Subfield(code='b', value='from journeyman to master /'),
        Subfield(code='c', value='Andrew Hunt, David Thomas.'),
    ]
)
record.add_field(field)
```

Or creating a record from a chunk of MARC in transmission format:

```python
record = Record(data=chunk)
```

Or getting a record as serialized MARC21.

```python
raw = record.as_marc()
```

You’ll normally want to use a MARCReader object to iterate through MARC records in a file.

### add_field(*fields)

Add pymarc.Field objects to a Record object.

Optionally you can pass in multiple fields.

### add_grouped_field(*fields) → None

Add pymarc.Field objects to a Record object and sort them “grouped”.

Which means, attempting to maintain a loose numeric order per the MARC standard for “Organization of the record” (http://www.loc.gov/marc/96principl.html). Optionally you can pass in multiple fields.

### add_ordered_field(*fields) → None

Add pymarc.Field objects to a Record object and sort them “ordered”.

Which means, attempting to maintain a strict numeric order. Optionally you can pass in multiple fields.

### addedentries

Returns Added entries fields.

Note: Fields 790-799 are considered “local” added entry fields but occur with some frequency in OCLC and RLIN records.

### as_dict() → Dict[str, str]

Turn a MARC record into a dictionary, which is used for as_json.
as_json(**kwargs) → str
   Serialize a record as JSON.

as_marc() → bytes
   Returns the record serialized as MARC21.

as_marc21() → bytes
   Returns the record serialized as MARC21.

author
   Returns the author from field 100, 110 or 111.

decode_marc(marc, to_unicode: bool = True, force_utf8: bool = False, hide_utf8_warnings: bool = False, utf8_handling: str = 'strict', encoding: str = 'iso8859-1') → None
   Populate the object based on the marc record in transmission format.
   The Record constructor actually uses decode_marc() behind the scenes when you pass in a chunk of MARC data to it.

fields

force_utf8

   Implements a dict-like get with a default value.
   If tag is not found, then the default value will be returned. The default value should be a Field instance.

get_fields(*args) → List[pymarc.field.Field]
   Return a list of all the fields in a record tags matching args.

    title = record.get_fields('245')

   If no fields with the specified tag are found then an empty list is returned. If you are interested in more than one tag you can pass it as multiple arguments.

    subjects = record.get_fields('600', '610', '650')

   If no tag is passed in to get_fields() a list of all the fields will be returned.

get_linked_fields(field: pymarc.field.Field) → List[pymarc.field.Field]
   Given a field that is not an 880, retrieve a list of any linked 880 fields.

isbn
   Returns the first ISBN in the record or None if one is not present.
   The returned ISBN will be all numeric, except for an x/X which may occur in the checksum position. Dashes and extraneous information will be automatically removed. If you need this information you’ll want to look directly at the 020 field, e.g. record['020']['a']. Values that do not match the regex will not be returned.

issn
   Returns the ISSN number [022]['a'] in the record or None.

issn_title
   Returns the key title of the record (222 $a and $b).

issnl
   Returns the ISSN-L number [022]['l'] of the record or None.

leader
location
   Returns location field (852).

notes
   Return notes fields (all 5xx fields).

physicaldescription
   Return physical description fields (300).

pos

publisher
   Return publisher from 260 or 264.
   Note: 264 field with second indicator ‘1’ indicates publisher.

pubyear
   Returns publication year from 260 or 264.

remove_field(*fields) → None
   Remove one or more pymarc.Field objects from a Record object.

remove_fields(*tags) → None
   Remove all the fields with the tags passed to the function.

   # remove all the fields marked with tags '200' or '899'.
   self.remove_fields('200', '899')

series
   Returns series fields.
   Note: 490 supersedes the 440 series statement which was both series statement and added entry. 8XX fields are added entries.

subjects
   Returns subjects fields.
   Note: Fields 690-699 are considered “local” added entry fields but occur with some frequency in OCLC and RLIN records.

sudoc
   Returns a Superintendent of Documents (SuDoc) classification number.
   Note: More information can be found at the following URL: https://www.fdlp.gov/classification-guidelines/introduction-to-the-classification-guidelines

title
   Returns the title of the record (245 $a and $b).

to_unicode

uniformtitle
   Returns the uniform title from field 130 or 240.

pymarc.record.map_marc8_record(record: pymarc.record.Record) → pymarc.record.Record
   Map MARC-8 record.

pymarc.record.normalize_subfield_code(subfield) → Tuple[Any, int]
   Normalize subfield code.
5.3 Writer

Pymarc Writer.

class pymarc.writer.JSONWriter(file_handle: IO)
    Bases: pymarc.writer.Writer
    A class for writing records as an array of MARC-in-JSON objects.
    IMPORTANT: You must the close a JSONWriter, otherwise you will not get valid JSON.
    Simple usage:

    .. code-block:: python

        from pymarc import JSONWriter
        # writing to a file
        writer = JSONWriter(open('file.json', 'wt'))
        writer.write(record)
        writer.close()  # Important!
        # writing to a string
        string = StringIO()
        writer = JSONWriter(string)
        writer.write(record)
        writer.close(close_fh=False)  # Important!
        print(string)

    close(close_fh: bool = True) → None
    Closes the writer.
    If close_fh is False close will also close the underlying file handle that was passed in to the constructor.
    The default is True.

    write(record: pymarc.record.Record) → None
    Writes a record.

class pymarc.writer.MARCWriter(file_handle: IO)
    Bases: pymarc.writer.Writer
    A class for writing MARC21 records in transmission format.
    Simple usage:

    .. code-block:: python

        from pymarc import MARCWriter
        # writing to a file
        writer = MARCWriter(open('file.dat', 'wb'))
        writer.write(record)
        writer.close()
        # writing to a string (Python 2 only)
        string = StringIO()
        writer = MARCWriter(string)
        writer.write(record)
        writer.close(close_fh=False)  # Important!
        print(string)
        # writing to memory (Python 3 only)
        memory = BytesIO()
        writer = MARCWriter(memory)
        writer.write(record)
        writer.close(close_fh=False)

    write(record: pymarc.record.Record) → None
    Writes a record.

class pymarc.writerTextWriter(file_handle: IO)
    Bases: pymarc.writer.Writer
    A class for writing records in prettified text MARCMaker format.
    A blank line separates each record.
Simple usage:

```python
from pymarc import TextWriter

# writing to a file
writer = TextWriter(open('file.txt', 'wt'))
writer.write(record)
writer.close()

# writing to a string
string = StringIO()
writer = TextWriter(string)
writer.write(record)
writer.close(close_fh=False)
print(string)
```

**write** *(record: pymarc.record.Record) → None*

Writes a record.

**class** pymarc.writer.Writer *(file_handle: IO)*

**Bases:** object

Base Writer object.

**close** *(close_fh: bool = True) → None*

Closes the writer.

If close_fh is False close will also close the underlying file handle that was passed in to the constructor.

The default is True.

**write** *(record: pymarc.record.Record) → None*

Write.

**class** pymarc.writer.XMLWriter *(file_handle: IO)*

**Bases:** pymarc.writer.Writer

A class for writing records as a MARCXML collection.

IMPORTANT: You must then close an XMLWriter, otherwise you will not get a valid XML document.

Simple usage:

```python
from pymarc import XMLWriter

# writing to a file
writer = XMLWriter(open('file.xml', 'wb'))
writer.write(record)
writer.close()  # Important!

# writing to a string (Python 2 only)
string = StringIO()
writer = XMLWriter(string)
writer.write(record)
writer.close(close_fh=False)  # Important!
print(string)

# writing to memory (Python 3 only)
memory = BytesIO()
writer = XMLWriter(memory)
writer.write(record)
writer.close(close_fh=False)  # Important!
```
close(close_fh: bool = True) → None
Closes the writer.
If close_fh is False close will also close the underlying file handle that was passed in to the constructor.
The default is True.
write(record: pymarc.record.Record) → None
Writes a record.

5.3.1 Field
The pymarc field file.

class pymarc.field.Field(tag: str, indicators: Optional[List[str]] = None, subfields: Optional[List[pymarc.field.Subfield]] = None, data: str = '')

Bases: object

Field() pass in the field tag, indicators and subfields for the tag.

```python
field = Field(
    tag = '245',
    indicators = ['0','1'],
    subfields = [
        Subfield(code='a', value='The pragmatic programmer : '),
        Subfield(code='b', value='from journeyman to master /'),
        Subfield(code='c', value='Andrew Hunt, David Thomas.'),
    ])
```

If you want to create a control field, don’t pass in the indicators and use a data parameter rather than a subfields parameter:

```python
field = Field(tag='001', data='fol05731351')
```

add_subfield(code: str, value: str, pos=None) → None
Adds a subfield code/value to the end of a field or at a position (pos).
If pos is not supplied or out of range, the subfield will be added at the end.

```python
field.add_subfield('u', 'http://www.loc.gov')
field.add_subfield('u', 'http://www.loc.gov', 0)
```

as_marc(encoding: str) → bytes
Used during conversion of a field to raw marc.

as_marc21(encoding: str) → bytes
Used during conversion of a field to raw marc.

classmethod convert_legacy_subfields(subfields: List[str]) → List[pymarc.field.Subfield]
Converts older-style subfield lists into Subfield lists.
Converts the old-style list of strings into a list of Subfields. As a class method this does not actually set any fields; it simply takes a list of strings and returns a list of Subfields.

```python
legacy_fields: list[str] = ['a', 'The pragmatic programmer : ',
    'b', 'from journeyman to master /',
    'c', 'Andrew Hunt, David Thomas']

coded_fields: list[Subfield] = Field.convert_legacy_subfields(legacy_fields)
```
myfield = Field(
    tag="245",
    indicators = ['0','1'],
    subfields=coded_fields
)

Parameters subfields – A list of [code, value, code, value]

Returns A list of Subfield named tuples

data

delete_subfield (code: str) → Optional[str]
    Deletes the first subfield with the specified ‘code’ and returns its value.

    value = field.delete_subfield('a')

    If no subfield is found with the specified code None is returned.

format_field () → str
    Returns the field as a string with tag, indicators, and subfield indicators.
    
    Like Field.value(), but prettier (adds spaces, formats subject headings).

get (code: str, default=None)
    A dict-like get method with a default value.

    Implements a non-raising getter for a subfield code that will return the value of the first subfield whose
    code is key.

get_subfields (*codes) → List[str]
    Get subfields matching codes.

    get_subfields() accepts one or more subfield codes and returns a list of subfield values. The order of the
    subfield values in the list will be the order that they appear in the field.

    print(field.get_subfields('a'))
    print(field.get_subfields('a', 'b', 'z'))

indicator1
    Indicator 1.

indicator2
    Indicator 2.

indicators

is_control_field () → bool
    Returns true or false if the field is considered a control field.
    
    Control fields lack indicators and subfields.

is_subject_field () → bool
    Returns True or False if the field is considered a subject field.

    Used by format_field().

linkage_occurrence_num () → Optional[str]
    Return the ‘occurrence number’ part of subfield 6, or None if not present.
subfields

subfields_as_dict() → Dict[str, List[T]]

Returns the subfields as a dictionary.

The dictionary is a mapping of subfield codes and values. Since subfield codes can repeat the values are a list.

tag

value() → str

Returns the field as a string w/ tag, indicators, and subfield indicators.

class pymarc.field.RawField(tag: str, indicators: Optional[List[str]] = None, subfields: Optional[List[pymarc.field.Subfield]] = None, data: str ="")

Bases: pymarc.field.Field

MARC field that keeps data in raw, un-decoded byte strings.

Should only be used when input records are wrongly encoded.

as_marc(encoding: Optional[str] = None)

Used during conversion of a field to raw MARC.

class pymarc.field.Subfield(code, value)

Bases: tuple

code

Alias for field number 0

value

Alias for field number 1

pymarc.field.map_marc8_field(f: pymarc.field.Field) → pymarc.field.Field

Map MARC8 field.

5.4 Exceptions

Exceptions for pymarc.

exception pymarc.exceptions.BadLeaderValue

Bases: pymarc.exceptions.PymarcException

Error when setting a leader value.

exception pymarc.exceptions.BadSubfieldCodeWarning

Bases: Warning

Warning about a non-ASCII subfield code.

exception pymarc.exceptions.BaseAddressInvalid

Bases: pymarc.exceptions.PymarcException

Base address exceeds size of record.

exception pymarc.exceptions.BaseAddressNotFound

Bases: pymarc.exceptions.PymarcException

Unable to locate base address of record.

exception pymarc.exceptions.EndOfRecordNotFound

Bases: pymarc.exceptions.FatalReaderError
Unable to locate end of record marker.

```python
exception pymarc.exceptions.FatalReaderError
   Bases: pymarc.exceptions.PymarcException
   Error preventing further reading.
```

```python
exception pymarc.exceptions.FieldNotFound
   Bases: pymarc.exceptions.PymarcException
   Record does not contain the specified field.
```

```python
exception pymarc.exceptions.MissingLinkedFields(field)
   Bases: pymarc.exceptions.PymarcException
   Error when a non-880 field has a subfield 6 that cannot be matched to an 880.
```

```python
exception pymarc.exceptions.NoActiveFile
   Bases: pymarc.exceptions.PymarcException
   There is no active file to write to in call to write.
```

```python
exception pymarc.exceptions.NoFieldsFound
   Bases: pymarc.exceptions.PymarcException
   Unable to locate fields in record data.
```

```python
exception pymarc.exceptions.PymarcException
   Bases: Exception
   Base pymarc Exception.
```

```python
exception pymarc.exceptions.RecordDirectoryInvalid
   Bases: pymarc.exceptions.PymarcException
   Invalid directory.
```

```python
exception pymarc.exceptions.RecordLeaderInvalid
   Bases: pymarc.exceptions.PymarcException
   Unable to extract record leader.
```

```python
exception pymarc.exceptions.RecordLengthInvalid
   Bases: pymarc.exceptions.FatalReaderError
   Invalid record length.
```

```python
exception pymarc.exceptions.TruncatedRecord
   Bases: pymarc.exceptions.FatalReaderError
   Truncated record data.
```

```python
exception pymarc.exceptions.WriteNeedsRecord
   Bases: pymarc.exceptions.PymarcException
   Write requires a pymarc.Record object as an argument.
```

## 5.5 MarcXML

From XML to MARC21 and back again.

```python
class pymarc.marcxml.XmlHandler(strict=False, normalize_form=None)
   Bases: xml.sax.handler.ContentHandler
```

### 5.5. MarcXML

From XML to MARC21 and back again.
XML Handler.

You can subclass XmlHandler and add your own process_record method that’ll be passed a pymarc.Record as it becomes available. This could be useful if you want to stream the records elsewhere (like to a rdbms) without having to store them all in memory.

**characters** *(chars)*

Append *chars* to *_text*.

**endElementNS** *(name, qname)*

End element NS.

**process_record** *(record)*

Append *record* to *records*.

**startElementNS** *(name, qname, attrs)*

Start element NS.

pymarc.marcxml.map_xml *(function, *files)*

Map a function onto the file.

So that for each record that is parsed the function will get called with the extracted record

```python
def do_it(r):
    print(r)
map_xml(do_it, 'marc.xml')
```

pymarc.marcxml.parse_xml *(xml_file, handler)*

Parse a file with a given subclass of xml.sax.handler.ContentHandler.

pymarc.marcxml.parse_xml_to_array *(xml_file, strict=False, normalize_form=None)*

Parse an XML file and return the records as an array.

Instead of passing in a file path you can also pass in an open file handle, or a file like object like StringIO. If you would like the parser to explicitly check the namespaces for the MARCSlim namespace use the strict=True option. Valid values for normalize_form are ‘NFC’, ‘NFKC’, ‘NFD’, and ‘NFKD’. See unicodedata.normalize for more info on these.

pymarc.marcxml.record_to_xml *(record, quiet=False, namespace=False)*

From MARC to XML.

pymarc.marcxml.record_to_xml_node *(record, quiet=False, namespace=False)*

Converts a record object to a chunk of XML.

If you would like to include the marcxml namespace in the root tag set namespace to True.

### 5.6 Constants

Constants for pymarc.

### 5.7 MARC-8

Handle MARC-8 files.

see [http://www.loc.gov/marc/specifications/speccharmarc8.html](http://www.loc.gov/marc/specifications/speccharmarc8.html)
class pymarc.marc8.MARC8ToUnicode(G0: int = 66, G1: int = 69, quiet: bool = False)
Bases: object
Converts MARC-8 to Unicode.

Note that currently, unicode strings aren’t normalized, and some codecs (e.g. iso8859-1) will fail on such strings. When I can require python 2.3, this will go away.

Warning: MARC-8 EACC (East Asian characters) makes some distinctions which aren’t captured in Unicode. The LC tables give the option of mapping such characters either to a Unicode private use area, or a substitute character which (usually) gives the sense. I’ve picked the second, so this means that the MARC data should be treated as primary and the Unicode data used for display purposes only. (If you know of either of fonts designed for use with LC’s private-use Unicode assignments, or of attempts to standardize Unicode characters to allow round-trips from EACC, or if you need the private-use Unicode character translations, please inform me, asl2@pobox.com.

ansel = 69
basic_latin = 66
translate(marc8_string)
Translate.

pymarc.marc8.marc8_to_unicode(marc8, hide_utf8_warnings: bool = False) \rightarrow str
Pass in a string, and get back a Unicode object.

```python
def marc8_to_unicode(record.title())
```

5.8 MARC-8 mapping

MARC-8 mapping.

5.8.1 Leader

The pymarc.leader file.

class pymarc.leader.Leader(leader: str)
Bases: object

Mutable leader.

A class to manipulate a Record’s leader.

You can use the properties (status, bibliographic_level, etc.) or their slices/index equivalent (leader[5], leader[7], etc.) to read and write values.

See LoC’s documentation for more infos about those fields.

```
leader = Leader("00475cas a2200169 i 4500")
leader[0:4] # returns "00475"
leader.status # returns "c"
leader.status = "a" # sets the status to "a"
leader[5] # returns the status "a"
leader[5] = "b" # sets the status to "b"
str(leader) # "00475bas a2200169 i 4500"
```

Usually the leader is accessed through the leader property of a record.

5.8. MARC-8 mapping
```
from pymarc import MARCReader
with open('test/marc.dat', 'rb') as fh:
    reader = MARCReader(fh)
    for record in reader:
        print(record.leader)
```

When creating/updating a `Record` please note that `record_length` and `base_address` will only be generated in the marc21 output of `record.as_marc()`

**base_address**
Base address of data (12-16).

**bibliographic_level**
Bibliographic level (07).

**cataloging_form**
Descriptive cataloging form (18).

**coding_scheme**
Character coding scheme (09).

**encoding_level**
Encoding level (17).

**implementation_defined_length**
Length of the implementation-defined portion (22).

**indicator_count**
Indicator count (10).

**length_of_field_length**
Length of the length-of-field portion (20).

**multipart_ressource**
Multipart resource record level (19).

**record_length**
Record length (00-04).

**record_status**
Record status (05).

**starting_character_position_length**
Length of the starting-character-position portion (21).

**subfield_code_count**
Subfield code count (11).

**type_of_control**
Type of control (08).

**type_of_record**
Type of record (06).
Indices and tables

• genindex
• modindex
• search
Python Module Index

p
pymarc.constants, 22
pymarc.exceptions, 20
pymarc.field, 18
pymarc.leader, 23
pymarc.marc8, 22
pymarc.marc8_mapping, 23
pymarc.marcxml, 21
pymarc.reader, 11
pymarc.record, 13
pymarc.writer, 16
Index

A
add_field() (pymarc.record.Record method), 13
add_grouped_field() (pymarc.record.Record method), 13
add_ordered_field() (pymarc.record.Record method), 13
add_subfield() (pymarc.field.Field method), 18
added_entries (pymarc.record.Record attribute), 13
ansel (pymarc.marc8.MARC8ToUnicode attribute), 23
as_dict() (pymarc.record.Record method), 13
as_json() (pymarc.record.Record method), 13
as_marc() (pymarc.field.Field method), 18
as_marc() (pymarc.field.RawField method), 20
as_marc() (pymarc.record.Record method), 14
as_marc21() (pymarc.field.Field method), 18
as_marc21() (pymarc.record.Record method), 14
author (pymarc.record.Record attribute), 14

B
BadLeaderValue, 20
BadSubfieldCodeWarning, 20
base_address (pymarc.leader.Leader attribute), 24
BaseAddressInvalid, 20
BaseAddressNotFoundException, 20
basic_latin (pymarc.marc8.MARC8ToUnicode attribute), 23
bibliographic_level (pymarc.leader.Leader attribute), 24

coding_scheme (pymarc.leader.Leader attribute), 24
current_chunk (pymarc.reader.MARCReader attribute), 12
current_exception (pymarc.reader.MARCReader attribute), 12
data (pymarc.field.Field attribute), 19
delete_marc() (pymarc.record.Record method), 14

E
encoding_level (pymarc.leader.Leader attribute), 24
endElementNS() (pymarc.marcxml.XmlHandler method), 22
EndOfRecordNotFound, 20
FatalReaderError, 21
Field (class in pymarc.field), 18
FieldNotFoundException, 21
fields (pymarc.record.Record attribute), 14
force_utf8 (pymarc.record.Record attribute), 14
format_field() (pymarc.field.Field attribute), 19

G
get() (pymarc.field.Field method), 19
get() (pymarc.record.Record method), 14
get_fields() (pymarc.record.Record method), 14
get_linked_fields() (pymarc.record.Record method), 14
get_subfields() (pymarc.field.Field method), 19

I
implementation_defined_length (pymarc.leader.Leader attribute), 24
map_xml() (in module pymarc.marcxml), 22
map_records() (in module pymarc.marcxml), 22
map_marc8_field() (in module pymarc.marc8), 20
map_marc8_record() (in module pymarc.marc8), 20
map_marc8_to_unicode() (in module pymarc.marc8), 23
map_records() (in module pymarc.marcxml), 15
map_xml() (in module pymarc.marcxml), 22
marc8_to_unicode() (in module pymarc.marc8), 23
MARC8ToUnicode (class in pymarc.marc8), 22
MARCReader (class in pymarc.reader), 11
MARCWriter (class in pymarc.writer), 16
MultipartResource, 21
MissingLinkedFields, 21
indicator1 (pymarc.field.Field attribute), 19
indicator2 (pymarc.field.Field attribute), 19
indicator_count (pymarc.leader.Leader attribute), 24
indicators (pymarc.field.Field attribute), 19
is_control_field() (pymarc.field.Field method), 19
is_subject_field() (pymarc.field.Field method), 19
isbn (pymarc.record.Record attribute), 14
issn (pymarc.record.Record attribute), 14
issn_title (pymarc.record.Record attribute), 14
issn1 (pymarc.record.Record attribute), 14
J
JSONReader (class in pymarc.reader), 11
JSONWriter (class in pymarc.writer), 16
L
Leader (class in pymarc.leader), 23
leader (pymarc.record.Record attribute), 14
length_of_field_length (pymarc.leader.Leader attribute), 24
linkage_occurrence_num() (pymarc.field.Field method), 19
location (pymarc.record.Record attribute), 14
M
map_marc8_field() (in module pymarc.marc8), 20
map_marc8_record() (in module pymarc.marcxml), 15
map_records() (in module pymarc.marcxml), 12
map_xml() (in module pymarc.marcxml), 22
marc8_to_unicode() (in module pymarc.marc8), 23
MARC8ToUnicode (class in pymarc.marc8), 22
MARCReader (class in pymarc.reader), 11
MARCWriter (class in pymarc.writer), 16
MissingLinkedFields, 21
multipart_resource (pymarc.leader.Leader attribute), 24
N
NoActiveFile, 21
NoFieldsFound, 21
normalize_subfield_code() (in module pymarc.record), 15
notes (pymarc.record.Record attribute), 15
P
parse_xml() (in module pymarc.marcxml), 22
parse_xml_to_array() (in module pymarc.marcxml), 22
physical_description (pymarc.record.Record attribute), 15
pos (pymarc.record.Record attribute), 15
process_record() (pymarc.marcxml.XmlHandler method), 22
publisher (pymarc.record.Record attribute), 15
pubyear (pymarc.record.Record attribute), 15
pymarc.constants (module), 22
pymarc.exceptions (module), 20
pymarc.field (module), 18
pymarc.leader (module), 23
pymarc.marc8 (module), 22
pymarc.marc8_mapping (module), 23
pymarc.marcxml (module), 21
pymarc.reader (module), 11
pymarc.record (module), 13
pymarc.writer (module), 16
Pymarexception, 21
R
RawField (class in pymarc.field), 20
Reader (class in pymarc.reader), 12
Record (class in pymarc.record), 13
record_length (pymarc.leader.Leader attribute), 24
record_status (pymarc.leader.Leader attribute), 24
record_to_xml() (in module pymarc.marcxml), 22
record_to_xml_node() (in module pymarc.marcxml), 22
RecordDirectoryInvalid, 21
RecordLeaderInvalid, 21
RecordLengthInvalid, 21
remove_field() (pymarc.record.Record method), 15
remove_fields() (pymarc.record.Record method), 15
S
series (pymarc.record.Record attribute), 15
startElementNS() (pymarc.marcxml.XmlHandler method), 22
starting_character_position_length
   (pymarc.leader.Leader attribute), 24
Subfield (class in pymarc.field), 20
subfield_code_count (pymarc.leader.Leader attribute), 24
subfields (pymarc.field.Field attribute), 19
subfields_as_dict() (pymarc.field.Field method), 19
subjects (pymarc.record.Record attribute), 15
sudoc (pymarc.record.Record attribute), 15
T
tag (pymarc.field.Field attribute), 20
TextWriter (class in pymarc.writer), 16
title (pymarc.record.Record attribute), 15
to_unicode (pymarc.record.Record attribute), 15
translate() (pymarc.marc8.MARC8ToUnicode method), 23
TruncatedRecord, 21
type_of_control (pymarc.leader.Leader attribute), 24
type_of_record (pymarc.leader.Leader attribute), 24

U
uniformtitle (pymarc.record.Record attribute), 15

V
value (pymarc.field.Subfield attribute), 20
value() (pymarc.field.Field method), 20

W
write() (pymarc.writer.JSONWriter method), 16
write() (pymarc.writer.MARCWriter method), 16
write() (pymarc.writer.TextWriter method), 17
write() (pymarc.writer.Writer method), 17
write() (pymarc.writer.XMLWriter method), 18
WriteNeedsRecord, 21
Writer (class in pymarc.writer), 17

X
XmlHandler (class in pymarc.marcxml), 21
XMLWriter (class in pymarc.writer), 17