Binary data structures (un-)Packing library

Release 1.2.0.dev1

Antonio Valentino

Dec 28, 2023

CONTENTS

1 Overview **3**
1.1 What is bpack? 3
1.2 Features 3
1.3 Limitations 4
1.4 Possible additional features still not implemented 5

2 Installation **7**
2.1 Pip 7
2.2 Conda 7
2.3 Testing 7

3 User Guide **9**
3.1 Core concepts 9
3.2 Binary data structures declaration 11
3.3 Fields specification 13
3.4 Enumeration fields 18
3.5 Sequence fields 20
3.6 Record nesting 21
3.7 Special type annotations 21
3.8 Data codecs 23

4 bpack package **25**
4.1 bpack.ba module 25
4.2 bpack.bs module 26
4.3 bpack.codecs module 27
4.4 bpack.descriptors module 28
4.5 bpack.enums module 32
4.6 bpack.np module 33
4.7 bpack.st module 36
4.8 bpack.typing module 37
4.9 bpack.utils module 38

5 Developers Guide **41**

5.1 Project links . . . 41
5.2 Set-up the development environment . . . 41
5.3 Testing the code . . . 42
5.4 Test coverage . . . 42
5.5 Check code style and formatting . . . 43
5.6 Build the documentation . . . 43
5.7 Test code snippets in the documentation . . . 43
5.8 Check documentation links . . . 43
5.9 Check documentation spelling . . . 44
5.10 Update the API documentation . . . 44

6 Copyright and License 45
6.1 Integral license text . . . 45

7 Release Notes 51
7.1 bpack v1.2.0 (UNRELEASED) . . . 51
7.2 bpack v1.1.0 (15/04/2023) . . . 51
7.3 bpack v1.0.0 (05/02/2023) . . . 52
7.4 bpack v0.8.2 (20/03/2022) . . . 52
7.5 bpack v0.8.1 (30/11/2021) . . . 52
7.6 bpack v0.8.0 (03/06/2021) . . . 52
7.7 bpack v0.7.1 (08/03/2021) . . . 52
7.8 bpack v0.7.0 (21/01/2021) . . . 53
7.9 bpack v0.6.0 (15/01/2021) . . . 53
7.10 bpack v0.5.0 (31/12/2020) . . . 54

Index 55

HomePage
https://github.com/avalentino/bpack

Author
Antonio Valentino

Contact
antonio.valentino@tiscali.it

Copyright
2020-2023, Antonio Valentino <antonio.valentino@tiscali.it>

Version
1.2.0.dev1

CHAPTER ONE

OVERVIEW

1.1 What is bpack?

The *bpack* Python package provides tools to describe and encode/decode binary data.

Binary data are assumed to be organized in *records*, each composed by a sequence of fields. Fields are characterized by a known size, offset (w.r.t. the beginning of the record) and datatype.

The package provides classes and functions that can be used to:

- describe binary data structures in a declarative way (structures can be specified up to the bit level)
- automatically generate encoders/decoders for a specified data descriptor

Encoders/decoders (*backends*) rely on well known Python packages like:

- `struct` (form the standard library)
- bitstruct (optional)
- numpy (optional)
- bitarray (optional) - partial implementation

1.2 Features

- declarative description of binary data structures
- specification of data structures up to *bit* level
- automatic *codec* generation from data descriptors
- decoding (from binary data to Python objects)
- encoding (from Python objects to binary data)
- backend:

  - *bpack.st* backend based on the `struct` module of standard Python library
  - *bpack.bs* backend based on bitstruct
  - *bpack.np* backend based on numpy
  - *bpack.ba* backend based on bitarray (only included for benchmarking purposes)
- support for signed/unsigned integer types
- support for `enum.Enum` types
- support for sequence types, i.e. fields with multiple (homogeneous) items
- both bit and byte order can be specified by the user
- automatic size determination for some data types
- record nesting (the field in a record descriptor can be another record)
- possibility to specify data types using the special type annotation class *bpack.typing.T* that accepts annotations and string specifiers compatible with the numpy "Array Interface" and `dtype`
- comprehensive test suite

1.3 Limitations

- only fixed size binary records are supported by design, the size of the record shall be known at the moment of the record descriptor definition. It should be easy for the user to leverage tools provided by the *bpack* Python package to support more complex decoding scenarios.
- currently it is assumed that all fields in a binary record share the same bit/byte order. The management of different byte order in the same binary record is, in principle, possible but not planned at the moment.
- sequence types can only contain basic numeric types; nested sequences, sequences of enums or sequences of records are not allowed at the moment.
- record nesting is only possible for records having the same base-units, bits or bytes, and compatible decoder types eventually.
- currently the *bpack.np* has a limited (incomplete) support to record nesting and encoding capabilities.

1.4 Possible additional features still not implemented

- user defined converters
- support for complex and datetime data types
- conversion to CSV, HDF5 and `numpy.dtype()`

CHAPTER

TWO

INSTALLATION

2.1 Pip

Basic installation:

```
$ python3 -m pip install bpack
```

Recommended:

```
$ python3 -m pip install bpack[bs,np]
```

to install also dependencies necessary to use the *bpack.bs* backend and (for binary structures defined up to the bit level).

2.2 Conda

```
$ conda install -c conda-forge -c avalentino bpack
```

2.3 Testing

To run the test suite it is necessary to have pytest installed:

```
$ python3 -m pytest --pyargs bpack
```

This only tests codec backends for which the necessary dependencies are available. To run a complete test please make sure to install all optional dependencies and testing libraries:

```
$ python3 -m pip install bpack[test]
```

CHAPTER

THREE

USER GUIDE

3.1 Core concepts

bpack is a lightweight Python package intended to help users to

- describe binary data structures
- encode/decode binary data to/from Python object

3.1.1 Descriptors

The user can define binary data structure in a declarative way, as follows:

```
import bpack

@bpack.descriptor
class BinaryRecord:
    field_1: float = bpack.field(size=8)
    field_2: int = bpack.field(size=4, signed=True)
```

Key concepts for definition of binary data structures are

- the declaration of the data structure by means of the *bpack.descriptors.descriptor()* class decorator. It allows to specify the main properties of the data structure.
- the specification of the characteristics of each field, mainly the data type, the size and (optionally) the offset with respect to the beginning of the record. This can be done using the *bpack.descriptors.field()* factory function.

In the above example the `BinaryRecord` has been defined to have two fields:

field_1
: a double precision floating point (8 bytes)

field_2
: a 32bit signed integer (4 bytes)

`size` is expressed in bytes in this case.

The offset of the fields have not been explicitly specified so they are computed automatically.

In the example `field_1` has `offset=0`, while `field_2` has `offset=8` i.e. data belonging to `field_2` immediately follow the ones of the previous field.

The design is strongly inspired to the one of the `dataclasses` package of the Python standard library.

3.1.2 Codecs

Once a binary structure is defined, the *bpack* package allows to automatically generate `Codec` objects that are able to convert binary data into a Python objects and vice versa:

```
import bpack.st

binary_data = b"\x18-DT\xfb!\t@\x15\xcd[\x07"

codec = bpack.st.Codec(BinaryRecord)
record = codec.decode(binary_data)

assert record.field_1 == 3.141592653589793
assert record.field_2 == 123456789

print(record)
```

```
BinaryRecord(field_1=3.141592653589793, field_2=123456789)
```

```
encoded_data = codec.encode(record)
assert binary_data == encoded_data

print("binary_data: ", binary_data)
print("encoded_data:", encoded_data)
```

```
binary_data:  b'\x18-DT\xfb!\t@\x15\xcd[\x07'
encoded_data: b'\x18-DT\xfb!\t@\x15\xcd[\x07'
```

In the example above it has been used the *`bpack.st.Codec`* class from the *`bpack.st`* module.

Please note that the decoder class (*`bpack.st.Codec`*)

- takes in input the *descriptor* (i.e. the type) of the binary data structure, and
- return a *codec* object which is capable to encode/decode only binary data organized according to the *descriptor* received at the instantiation time. If one need to encode/decode a differed

data structure than it is necessary to instantiate a different codec.

The `bpack.st` module used in the example is just one of the, so called, *backends* available in *bpack*.

See the *Backends* section below for more details.

3.2 Binary data structures declaration

As anticipated above the declaration of a binary data structure and its main properties is done using the `bpack.descriptors.descriptor()` class decorator.

3.2.1 Bit vs byte structures

One of the properties that the `bpack.descriptors.descriptor()` class decorator allows to specify is *baseunits*. It allows to specify the elementary units used to describe the binary structure itself. A structure can be described in terms of *bytes* or in terms of *bits*, i.e. if field size and offsets have to be intended as number of bytes of as number of bits.

This is an important distinction for two reasons:

- it is fundamental for *decoders* (see below) to know much data have to be converted and where this data are exactly located in a string of bytes
- not all *backends* are capable of decoding both kinds of structures

Note: Currently available *backends* do not support nested data structures (see *Record nesting*) described using different *baseunits* (see *Limitations*).

Baseunits can be specified as follows:

```
@bpack.descriptor(baseunits=bpack.EBaseUnits.BITS)
class BitRecord:
    field_1: bool = bpack.field(size=1)
    field_2: int = bpack.field(size=3)
    field_3: int = bpack.field(size=4)
```

The `baseunits` parameter has been specified as a parameter of the `bpack.descriptors.descriptor()` class decorator and its possible values are enumerated by the `bpack.enums.EBaseUnits` `enum.Enum`:

- `bpack.enums.EBaseUnits.BITS`, or
- `bpack.enums.EBaseUnits.BYTES`

If the `baseunits` parameter is not specified than it is assumed to be equal to *bpack.enums.EBaseUnits.BYTES* by default.

Please note that the entire data structure of the above example is only 8 bits (1 byte) large.

Note: Please note that *baseunits* and many of the function and method parameters whose valued is supposed to be an `enum.Enum` can also accept a string value. E.g. the above example can also be written as follows:

```
@bpack.descriptor(baseunits="bits")
class BitRecord:
    field_1: bool = bpack.field(size=1)
    field_2: int = bpack.field(size=3)
    field_3: int = bpack.field(size=4)
```

Please refer to the specific enum documentation (in this case *bpack.enums.EBaseUnits*) to know which are string values corresponding to the desired enumerated value.

3.2.2 Specifying bit/byte order

Other important parameters for the *bpack.descriptors.descriptor()* class decorator are:

byteorder
whose possible values are described by *bpack.enums.EByteOrder*. By the fault the native byte order is assumed.

bitorder
whose possible values are described by *bpack.enums.EBitOrder*. The *bitorder* parameter shall always be set to `None` the if *baseunits* value is *bpack.enums.EBaseUnits.BYTES*.

Both this parameters describe the internal organization of binary data of each field.

3.2.3 Descriptor size

The *bpack.descriptors.descriptor()* class decorator also allows to specify *explicitly* the overall size of the binary data structure:

```
@bpack.descriptor(baseunits="bits", size=8)
class BinaryRecord:
    field_1: bool = bpack.field(size=1)
    field_2: int = bpack.field(size=3)
```

In this case the the overall size of `BitRecord` is 8 bits (1 bytes)

```
>>> bpack.calcsize(BinaryRecord)
8
```

even if the sum of sizes of all fields is only 4 bits.

Usually explicitly specifying the *size* of a binary data structure is not necessary because the *bpack* is able to compute it automatically by looking at the size of fields.

In some cases, anyway, it can be useful to specify it, e.g. when one want to use a descriptor like the one defined in the above example as field of a larger descriptor (see *Record nesting*). In this case it is important tho know the correct size of each field in order to be able to automatically compute the *offset* of the following ones.

3.3 Fields specification

As anticipated in the previous section there are three main elements that the *bpack* package need to know about fields in order to have a complete description of a binary data structure:

- the field data **type**,
- the field **size** (expressed in *baseunits*, see *Bit vs byte structures*), and
- the field **offset** with respect to the beginning of the binary data structure (also in this case expressed in *baseunits*, see *Bit vs byte structures*)

```
@bpack.descriptor
class BinaryRecord:
    field: int = bpack.field(size=4, offset=0)
```

Please note, anyway, that in some case it is possible to infer some of the above information from the context so it is not always necessary to specify all of them explicitly. More details will be provided in the following.

As shown in the example above the main way to specify a field descriptor is to use the `bpack.descriptors.field()` factory function together with Python type annotations to specify the data type.

3.3.1 Type

The data type of a field is the only parameter that is always mandatory, and also it is the only parameter that is not specified by means of the *bpack.descriptors.field()* factory function. Rather it is specified using the standard Python syntax for type annotations.

Currently supported data types are:

basic types
: basic Python types like `bool`, `int`, `float`, `bytes`, `str` (`complex` is not supported currently)

enums
: enumeration types defined using the `enum` module of the standard Python library. Please refer to the *Enumeration fields* section for more details about features and limitations

sequences
: used to define fields containing a sequence of homogeneous values (i.e. values having the same data type). A *sequence* data type in *bpack* can be defined using the standard type annotations classes like `typing.Sequence` or `typing.List`. Please refer to the *Sequence fields* section for more details about features and limitations

descriptors
: i.e. any binary data structure defined using the *bpack.descriptors.descriptor()* class decorator (see also *Record nesting*)

type annotations
: annotated data types defined by means of the *bpack.typing.T* type annotation. Please refer to the *Special type annotations* section for a more detailed description

---

Note: The `str` type in Python is used to represent unicode strings. The conversion of this kind of strings from/to binary format requires some form of decoding/encoding. *Bpack* codecs (see *Data codecs*) convert `str` data from/to `bytes` strings using the "UTF-8" encoding.

Please note that the *size* of a `str` field still describes the number of bits/bytes in its binary representation, not the length of the string (which in principle could require a number of bytes larger that the number of characters).

3.3.2 Size

The field *size* is specified as a positive integer in *baseunits* (see the *Bit vs byte structures* section).

It is a fundamental information and it must be always specified by means of the `bpack.descriptors.field()` factory function unless it is absolutely clear and unambiguous how to determine the fields size from the data type.

This is only possible in the following cases:

- the data type is `bool` in which case the size is assumed to be 1 (at the moment no other basic type has a default size associated)
- the data type is a record descriptor, in which case the field size is computed as follows:

```
bpack.calcsize(BinaryRecord, units=bpack.baseunits(BinaryRecord))
```

- the data type is specified using special type annotations also including size information:

```
from bpack import T

@bpack.descriptor
class BinaryRecord:
    field: T["u3"]
```

  The `T["u3"]` type annotation specifier defines an unsigned integer type (`u`) having size 3 (for the specific example this means 3 bytes). Please refer to the *Special type annotations* section for more details.

Please note that the size of the field must not necessarily correspond to the size of one of the data types supported by the platform. In the example above it has been specified a type `T["u3"]` which corresponds to a 24 bits unsigned integer. It is represented using a standard Python `int` in the Python code but the binary representation will always take only 3 bytes.

3.3.3 Offset

The field *offset* is specified as a not-negative integer in *baseunits* (see the *Bit vs byte structures* section), and it represent the amount of *baseunits* from the beginning of the record to the beginning of the field.

It is a fundamental information and it can be specified by means of the `bpack.descriptors.field()` factory function.

The *bpack* package, anyway, implements a mechanism to automatically compute the field offset exploiting information of the other fields in the record. For this reason it is necessary to specify the field *offset* explicitly only in very specific cases.

For example the *verbose* definition of a record with 5 integer fields looks like the following:

```
@bpack.descriptor
class BinaryRecord:
    field_1: int = bpack.field(size=4, offset=0)
    field_2: int = bpack.field(size=4, offset=4)
    field_3: int = bpack.field(size=4, offset=8)
    field_4: int = bpack.field(size=4, offset=12)
    field_5: int = bpack.field(size=4, offset=16)
```

If not specified, the offset of the first field is assumed to be `0`, and the offset of the following fields is assumed to be equal to the offset of the previous field plus the size of the previous field itself:

```
field[n].offset = field[n - 1].offset + field[n - 1].size
```

In short the automatic offset computation works assuming that all fields are stored contiguously and without holes.

```
@bpack.descriptor
class BinaryRecord:
    field_1: int = bpack.field(size=4)  # offset = 0 first field
    field_2: int = bpack.field(size=4)  # offset = 4
                                        # field_1.offset + field_1.size
    field_3: int = bpack.field(size=4)  # offset = 8
                                        # field_2.offset + field_2.size
    field_4: int = bpack.field(size=4)  # offset = 12
                                        # field_3.offset + field_3.size
    field_5: int = bpack.field(size=4)  # offset = 16
                                        # field_4.offset + field_4.size
```

Now suppose that the user is not interested in the field n. 2 and wants to remove it from the descriptor. This creates a *gap* in the binary data which makes not possible to exploit the automatic offset computation mechanism:

```
@bpack.descriptor
class BinaryRecord:
    field_1: int = bpack.field(size=4)    # offset = 0 first field
    # field_2: int = bpack.field(size=4)
    field_3: int = bpack.field(size=4)    # offset = 4 != 8   NOT CORRECT
    field_4: int = bpack.field(size=4)    # offset = 8 != 12  NOT CORRECT
    field_5: int = bpack.field(size=4)    # offset = 12 != 16 NOT CORRECT
```

The automatic computation of the offset fails, in this case, because of the missing information about `field_2`. Indeed, since `field_2` has not been specified, for the computation of the offset of `field_3` *bpack* assumes that the previous field is `field_1` and performs the computation accordingly:

```
field_3.offest = fielf_1.offset + field_i.size == 4 != 8  # INCORRECT
```

The incorrect offset of `field_3` causes the incorrect computation of the offset all the fields that follow.

One option to recover the correct behavior (without falling back to the *verbose* description shown at the beginning of the section) is to specify explicitly **only** the offset of the first field after the gap:

```
@bpack.descriptor
class BinaryRecord:
    field_1: int = bpack.field(size=4)    # offset = 0 first field
    # field_2: int = bpack.field(size=4)
    field_3: int = bpack.field(size=4, offset=8)
    field_4: int = bpack.field(size=4)    # offset = 12
    field_5: int = bpack.field(size=4)    # offset = 16
```

In this way the correct offset can be computed automatically for all fields but the one(s) immediately following a *gap* in the data descriptor.

3.3.4 Signed integer types

Only for integer types, it is possible to specify if the integer value is *signed* or not. Although this distinction is not relevant in the Python code, it is necessary to have this information when data have to be stored in binary form.

```
@bpack.descriptor
class BinaryRecord:
    field: int = bpack.field(size=4, offset=0, signed=True)
```

If *signed* is not specified for a field having an integer type, then it is assumed to be `False` (*unsigned*).

The *signed* parameter is ignored if the data type is not `int`.

3.3.5 Default values

The *bpack.descriptors.field()* factory function also allows to specify default values using the `default` parameter:

```
@bpack.descriptor
class BinaryRecord:
    field: int = bpack.field(size=4, default=0)
```

This allows to instantiate the record without specifying the value of each field:

```
>>> BinaryRecord()
BinaryRecord(field=0)
```

In cases in which the *bpack.descriptors.field()* factory function is not used for field definition, the default value can be specified by direct assignment:

```
@bpack.descriptor
class BinaryRecord:
    field_1: bool = False
    field_2: bpack.T["i4"] = 33
```

Note: No check is performed by *bpack* to ensure that the default value specified for a field is consistent with the corresponding data type.

3.4 Enumeration fields

The *bpack* package supports direct mapping of integer types, strings of `bytes` and Python `str` (unicode) into enumerated values of Python `Enum` types (including also `IntEnum` and `IntFlag`).

Example:

```
import enum

class EColor(enum.IntEnum):
    RED = 1
    GREEN = 2
    BLUE = 3
    BLACK = 10
    WHITE = 11

@bpack.descriptor(baseunits="bits")
class BinaryRecord:
    foreground: EColor = bpack.field(size=4, default=EColor.BLACK)
    background: EColor = bpack.field(size=4, default=EColor.WHITE)

record = BinaryRecord()
print(record)
```

```
BinaryRecord(foreground=<EColor.BLACK: 10>, background=<EColor.WHITE: 11>)
```

The `EColor` enum values are lower that 16 so they can be represented with only 4 bits.

In particular the binary representation of `BLACK` and `WHITE` is:

```
>>> format(EColor.BLACK, "04b")
'1010'
>>> format(EColor.WHITE, "04b")
'1011'
```

and the binary string representing the above defined binary record is:

```
data = bytes([0b10101011])
print(data)
```

```
b'\xab'
```

The data string can be decoded using the *bpack.bs* backend that is suitable to handle binary data structures with `bits` as *baseunits*:

```
import bpack.bs

decoder = bpack.bs.Decoder(BinaryRecord)
record = decoder.decode(data)
print(record)
```

```
BinaryRecord(foreground=<EColor.BLACK: 10>, background=<EColor.WHITE: 11>)
```

The result is directly mapped into Python enum values: `EColor:BLACK` and `EColor:WHITE`.

Note: The `Enum` sub-classes are accepted as field type only if all the enumeration values have the same type (`int`, `bytes` or `str`).

Example:

```
import enum
import bpack


class EType(enum.Enum):
    A = "A"
    B = 2


@bpack.descriptor(baseunits=bpack.EBaseUnits.BITS)
class Record:
    field: EType = bpack.field(size=8, default=EType.A)   # ERROR!
```

The above code will result in the following error:

```
1 @bpack.descriptor(baseunits=bpack.EBaseUnits.BITS)
2 class Record:
3     field: EType = bpack.field(size=8, default=EType.A)

[...]

TypeError: only Enum with homogeneous values are supported
```

3.5 Sequence fields

bpack provides a basic support to homogeneous *sequence* fields i.e. fields containing a sequence of values having the same data type.

The sequence is specified using the standard Python type annotation classes `typing.Sequence` or `typing.List`.

The data type of a sequence item can be any of the basic data types described in *Type*.

```
from typing import Sequence, List

@bpack.descriptor
class BinaryRecord:
    sequence: Sequence[int] = bpack.field(size=1, repeat=2)
    list: List[float] = bpack.field(size=4, repeat=3)
```

Please note that the *size* parameter of the `bpack.descriptors.field()` factory function describes the size of the sequence *item*, while the *repeat* parameter described the number of elements in the *sequence*.

The `bpack.bs` and `bpack.st` backend map `Sequence[T]` onto Python `tuple` instances and `List[T]` onto `list` instances. The `bpack.np` instead maps all kind of sequences onto `numpy.ndarray` instances.

3.6 Record nesting

Descriptors of binary structures (record types) can have fields that are binary structure descriptors in their turn (sub-records).

Example:

```
@bpack.descriptor
class SubRecord:
    field_21: int = bpack.field(size=2, default=1)
    field_22: int = bpack.field(size=2, default=2)

@bpack.descriptor
class Record:
    field_1: int = bpack.field(size=4, default=0)
    field_2: SubRecord = bpack.field(default_factory=SubRecord)

print(Record())
```

```
Record(field_1=0, field_2=SubRecord(field_21=1, field_22=2))
```

Decoding of the `Record` structure will automatically decode also data belonging to the sub-record and assign to `field_2` a `SubRecord` instance.

3.7 Special type annotations

Using the *bpack.descriptors.field()* factory function to define fields can be sometime very verbose and boring.

The *bpack* package provides a typing annotation helper, *bpack.typing.T*, that allows to specify basic types annotated with additional information like the *size* or the *signed* attribute for integers. This helps to reduce the amount of typesetting required to specify a binary structure.

The *bpack.typing.T* type annotation class takes in input a string argument and converts it into an annotated basic type.

```
>>> T["u4"]
typing.Annotated[int, TypeParams(byteorder=None, type='int',
                                 size=4, signed=False)]
```

The resulting type annotation is a `typing.Annotated` basic type with attached a *bpack.typing.TypeParams* instance.

This allows *bpack* to retrieve the information necessary to specify a field.

For example the following descriptor:

```
@bpack.descriptor
class BinaryRecord:
    field_1: int = bpack.field(size=4, signed=True, default=0)
    field_2: int = bpack.field(size=4, signed=False, default=1)
```

Can be specified in a more synthetic form as follows:

```
@bpack.descriptor
class BinaryRecord:
    field_1: T["i4"] = 0
    field_2: T["u4"] = 1
```

String descriptors, or *typestr*, are compatible with numpy (a sub-set of the one used in the numpy array interface).

The *typestr* string format consists of 3 parts:

- an (optional) character describing the bit/byte order of the data
  - `<`: little-endian,
  - `>`: big-endian,
  - `|`: not-relevant
- a character code giving the basic type of the array, and
- an integer providing the number of bits/bytes used by the type

The basic type character codes are:

- `i`: sighed integer
- `u`: unsigned integer
- `f`: float
- `c`: complex (**currently not supported**)
- `S`: bytes (string)

Note: Although the *typestr* format allows to specify the bit/byte *order* of the datatype it is usually not necessary to do it because descriptor object already have this information.

See also:

`bpack.typing.str_to_type_params()`, *bpack.typing.TypeParams*, https://numpy.org/doc/stable/reference/arrays.dtypes.html and https://numpy.org/doc/stable/reference/arrays.interface.html

3.8 Data codecs

3.8.1 Backends

Backends provide encoding/decoding capabilities for binary data *descriptors* exploiting external packages to do the low level job.

Currently *bpack* provides the following backends:

- `bpack.st` backend, based on the `struct` package, and
- `bpack.bs` backend, based on the bitstruct package to decode binary data described at bit level, i.e. with fields that can have size expressed in terms of number of bits (also smaller that 8).
- `bpack.np` backend, based on numpy (limited encoding capabilities)

Additionally a `bpack.ba` backend, feature incomplete, is also provided mainly for benchmarking purposes. The `bpack.ba` backend is based on the bitarray package.

3.8.2 Codec objects

Each backend provides a `Codec` class that can be used to instantiate a *codec* objects.

Please refer to the *Codecs* section for a description of basic concepts of how decoders work.

Decoders are instantiated passing to the `Codec` class a binary data record *descriptor*. Each *codec* has

- a `descriptor` property, by which it is possible to access the *descriptor* associated to the `Codec` instance
- a `baseunits` property, that indicates the kind of *descriptors* supported by the `Decoder` class
- a `decode(data: bytes)` method, that takes in input a string of `bytes` and returns an instance of the record type specified at the instantiation of the *codec* object
- a `encode(record)` method, that takes in input an instance of the record type specified at the instantiation of the *codec* object (a Python object) and returns a string of `bytes`

Details on the `Codec` API can be found in:

- `bpack.bs.Codec`,
- `bpack.np.Codec`,
- `bpack.st.Codec`

Note: the *bpack.ba* backend does not provides encoding capabilities so no `bpack.ba.Codec` class exists. A *bpack.ba.Decoder* class exists instead providing only decoding capabilities.

3.8.3 Codec decorator

Each backend provides also a `@codec` decorator the can be used to add to a *descriptor* direct decoding capabilities. In particular the `frombytes(data: bytes)` class method and the `tobytes()` method are added to the *descriptor* to be able to write code as the following:

```
import bpack
import bpack.st

@bpack.st.codec
@bpack.descriptor
class BinaryRecord:
    field_1: float = bpack.field(size=8)
    field_2: int = bpack.field(size=4, signed=True)

binary_data = b"\x18-DT\xfb!\t@\x15\xcd[\x07"
record = BinaryRecord.frombytes(binary_data)

print(record)
```

```
BinaryRecord(field_1=3.141592653589793, field_2=123456789)
```

```
encoded_data = record.tobytes()
assert binary_data == encoded_data

print(encoded_data)
```

```
b'\x18-DT\xfb!\t@\x15\xcd[\x07'
```

CHAPTER FOUR

BPACK PACKAGE

Binary data structures (un-)Packing library.

bpack provides tools to describe, in a *declarative* way, and encode/decode binary data.

4.1 bpack.ba module

Bitarray based codec for binary data structures.

class `bpack.ba.Decoder`(*descriptor*, *converters=<function converter_factory>*)

Bases: `Decoder`

Bitarray based data decoder.

Only supports "big endian" byte-order and MSB bit-order.

decode(*data: bytes*)

Decode binary data and return a record object.

baseunits: *EBaseUnits* = 'bits'

`bpack.ba.decoder`(*cls*)

Class decorator to add (de)coding methods to a descriptor class.

The decorator automatically generates a *Codec* object form the input descriptor class and attach to it methods for conversion form/to bytes.

4.2 bpack.bs module

Bitstruct based codec for binary data structures.

class bpack.bs.**Codec**(*descriptor*, *codec=None*, *decode_converters=None*, *encode_converters=None*)

Bases: BaseStructCodec

Bitstruct based codec.

Default bit-order: MSB.

baseunits: *EBaseUnits* **= 'bits'**

bpack.bs.**Decoder**

alias of *Codec*

bpack.bs.**Encoder**

alias of *Codec*

bpack.bs.**codec**(*cls*)

Class decorator to add (de)coding methods to a descriptor class.

The decorator automatically generates a *Codec* object form the input descriptor class and attach to it methods for conversion form/to bytes.

bpack.bs.**decoder**(*cls*)

Class decorator to add (de)coding methods to a descriptor class.

The decorator automatically generates a *Codec* object form the input descriptor class and attach to it methods for conversion form/to bytes.

bpack.bs.**encoder**(*cls*)

Class decorator to add (de)coding methods to a descriptor class.

The decorator automatically generates a *Codec* object form the input descriptor class and attach to it methods for conversion form/to bytes.

bpack.bs.**packbits**(*values*, *bits_per_sample: int*, *signed: bool = False*, *byteorder: str = ''*) → bytes

Pack integer values using the specified number of bits for each sample.

Converts a sequence of values into a string of bytes in which each sample is stored according to the specified number of bits.

Example:

```
            4 samples                                   3 bytes
```

(continues on next page)

(continued from previous page)

```
[samp_1, samp_2, samp_3, samp_4] --> |------|------|------|------|

                                      4 samples (6 bits per sample)
```

Please note that no check that the input values actually fits in the specified number of bits is performed is performed.

The function return a sting of bytes including same number of samples of the input plus possibly some padding bit (at the end) to fill an integer number of bytes.

If `signed` is set to True integers are stored as signed integers.

`bpack.bs.unpackbits`(*data: bytes*, *bits_per_sample: int*, *signed: bool = False*, *byteorder: str = ''*)

Unpack packed (integer) values form a string of bytes.

Takes in input a string of bytes in which (integer) samples have been stored using `bits_per_sample` bit for each sample, and returns the sequence of corresponding Python integers.

Example:

```
         3 bytes                            4 samples

|------|------|------|------| --> [samp_1, samp_2, samp_3, samp_4]

4 samples (6 bits per sample)
```

If `signed` is set to True integers are assumed to be stored as signed integers.

4.3 bpack.codecs module

Base classes and utility functions for codecs.

`class bpack.codecs.Codec`(*descriptor*)

Bases: `Decoder`, `Encoder`, `ABC`

Base class for codecs.

`class bpack.codecs.Decoder`(*descriptor*)

Bases: `BaseCodec`, `ABC`

Base class for decoders.

`abstract decode`(*data: bytes*)

Decode binary data and return Python object.

class bpack.codecs.**Encoder**(*descriptor*)

Bases: `BaseCodec`, `ABC`

Base class for encoders.

abstract encode(*record*) → bytes

Encode python objects into binary data.

bpack.codecs.**has_codec**(*descriptor*, *codec_type: Type[*Decoder | Encoder | Codec*] | None = None*) → bool

Return True if the input descriptor has a codec attached.

A descriptor decorated with a *codec* decorator has an attached codec instance and "from-bytes"/"tobytes" methods (depending on the kind of codec).

The *codec_type* parameter can be used to query for specific codec features:

- codec_type = None: return True for any kind of codec
- codec_type = *Decoder*: return True if the attached coded has decoding capabilities
- codec_type = *Encoder*: return True if the attached coded has encoding capabilities
- codec_type = *Codec*: return True if the attached coded has both encoding and decoding capabilities

4.4 bpack.descriptors module

Descriptors for binary records.

class bpack.descriptors.**BinFieldDescriptor**(*type: Type | None = None*, *size: int | None = None*, *offset: int | None = None*, *signed: bool | None = None*, *repeat: int | None = None*)

Bases: `object`

Descriptor for bpack fields.

See also:

bpack.descriptors.field() for a description of the attributes.

is_enum_type() → bool

Return True if the field is an enum.

is_int_type() → bool

Return True if the field is an integer or a sub-type of integer.

is_sequence_type() → bool

Return True if the field is a sequence.

update_from_type(*type_: Type*)

Update the field descriptor according to the specified type.

validate()

Perform validity check on the BinFieldDescriptor instance.

offset: int | None = None

repeat: int | None = None

number of items

signed: bool | None = None

size: int | None = None

item size

property total_size

Total size in bytes of the field (considering all item).

type: Type | None = None

class bpack.descriptors.**Field**(*default*, *default_factory*, *init*, *repr*, *hash*, *compare*, *metadata*, *kw_only*)

Bases: object

compare

default

default_factory

hash

init

kw_only

metadata

name

repr

type

`bpack.descriptors.asdict(obj, *, dict_factory=<class 'dict'>) → dict`

Return the fields of a record as a new dictionary.

The returned dictionary maps field names to field values.

If given, 'dict_factory' will be used instead of built-in dict. The function applies recursively to field values that are dataclass instances. This will also look into built-in containers: tuples, lists, and dicts.

`bpack.descriptors.astuple(obj, *, tuple_factory=<class 'tuple'>) → Sequence`

Return the fields of a dataclass instance as new tuple of field values.

If given, 'tuple_factory' will be used instead of built-in tuple. The function applies recursively to field values that are dataclass instances. This will also look into built-in containers: tuples, lists, and dicts.

`bpack.descriptors.baseunits(obj) → EBaseUnits`

Return the base units of a binary record descriptor.

`bpack.descriptors.bitorder(obj) → EBitOrder | None`

Return the bit order of a binary record descriptor.

`bpack.descriptors.byteorder(obj) → EByteOrder`

Return the byte order of a binary record descriptor (endianness).

`bpack.descriptors.calcsize(obj, units: EBaseUnits | None = None) → int`

Return the size of the `obj` record.

If the *units* parameter is not specified (default) then the returned *size* is expressed in the same *base units* of the descriptor.

`bpack.descriptors.descriptor(cls, *, size: int | None = None, byteorder: str | EByteOrder = EByteOrder.DEFAULT, bitorder: str | EBitOrder | None = None, baseunits: EBaseUnits = EBaseUnits.BYTES, **kwargs)`

Class decorator to define descriptors for binary records.

It converts a dataclass into a descriptor object for binary records.

- ensures that all fields are `bpack.descriptor.Field` descriptors
- offsets are automatically computed if necessary
- consistency checks on offsets and sizes are performed

Parameters

- **cls** – class to be decorated
- **size** – the size (expressed in *base units*) of the binary record
- **byteorder** – the byte-order of the binary record

- **bitorder** – the bit-order of the binary record (must be None if the *base units* are bytes). If set to none in bit-based records it is assumed *bpack.enums.EBitOrder.DEFAULT* which corresponds to *bpack.enums.EBitOrder.MSB* in all decoders currently implemented.
- **baseunits** – the base units (*bpack.enums.EBaseUnits.BITS* or *bpack.enums.EBaseUnits.BYTES*) used to specify the binary record descriptor

It is also possible to specify as additional keyword arguments all the parameters accepted by dataclasses.dataclass().

`bpack.descriptors.`**`field`**`(*, size: int | None = None, offset: int | None = None, signed: bool | None = None, repeat: int | None = None, metadata=None, **kwargs) → Field`

Initialize a field descriptor.

Returned object is a *Field* instance with metadata properly initialized to describe the field of a binary record.

Parameters

- **size** – int size of the field in *bpack.enums.EBaseUnits*
- **offset** – int offset of the field w.r.t. the beginning of the record (expressed in *bpack.enums.EBaseUnits*)
- **signed** – bool True if an *int* field is signed, False otherwise. This parameter must not be specified for non *int* fields.
- **repeat** – int length of the sequence for *sequence* fields, i.e. fields consisting in multiple items having the same data type. This parameter must not be specified if the data type is not a sequence type (e.g. typing.List).
- **metadata** – additional metadata to be attached the the field descriptor.
- **kwargs** – additional keyword arguments for the dataclasses.field() function.

`bpack.descriptors.`**`field_descriptors`**`(descriptor, pad: bool = False) → Iterator[BinFieldDescriptor]`

Iterate the input record descriptor and return binary field descriptors.

Returned items are instances of the *BinFieldDescriptor* class describing characteristics of each field of the input binary record descriptor.

If the pad parameter is set to True then also generate dummy field descriptors for padding elements necessary to take into account offsets between fields.

`bpack.descriptors.`**`fields`**`(obj) → Sequence[Field]`

Return a tuple describing the fields of this descriptor.

`bpack.descriptors.`**`flat_fields_iterator`**(*descriptor*, *offset: int = 0*) → Iterator[*Field*]

Recursively iterate on fields of a record descriptor.

The behavior of this function is similar to the one of *`bpack.descriptors.fields()`* if the input descriptor do not contain fields that are descriptors (nested). The main difference is that this one is an iterator while *`bpack.descriptors.fields()`* returns a tuple.

If the input descriptor is nested (i.e. has fields that are descriptors), then a the it is visited recursively to return all the fields belonging to the main descriptor and to the nested ones.

The nested descriptors are replaced by their fields and the returned sequence of fields is *flat*.

Note: please note that in case of nested descriptors, the returned fields are copy of the original ones, with the *offset* attribute adjusted to the relative to the beginning of the root descriptor.

`bpack.descriptors.`**`get_field_descriptor`**(*field: Field*, *validate: bool = True*) → *BinFieldDescriptor*

Return the field descriptor attached to a *`Field`*.

`bpack.descriptors.`**`is_descriptor`**(*obj*) → bool

Return true if `obj` is a descriptor or a descriptor instance.

`bpack.descriptors.`**`is_field`**(*obj*) → bool

Return true if an `obj` can be considered is a field descriptor.

`bpack.descriptors.`**`set_field_descriptor`**(*field: Field*, *descriptor: BinFieldDescriptor*, *validate: bool = True*) → *Field*

Set the field metadata according to the specified descriptor.

4.5 bpack.enums module

Enumeration types for the bpack package.

`class` `bpack.enums.`**`EBaseUnits`**(*value*, *names=None*, *, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: `Enum`

Base units used to specify size and offset parameters in descriptors.

`BITS = 'bits'`

`BYTES = 'bytes'`

class bpack.enums.EBitOrder(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: `Enum`

Enumeration for bit order.

DEFAULT = ''

LSB = '<'

MSB = '>'

class bpack.enums.EByteOrder(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: `Enum`

Enumeration for byte order (endianness).

Note: the *EByteOrder.DEFAULT* is equivalent to *EByteOrder.NATIVE* for binary structures having `EBaseUnits.BYTE` base units, and *EByteOrder.BE* for binary structures having `EBaseUnits.BIT` base units.

classmethod get_native()

Return the native byte order.

BE = '>'

DEFAULT = ''

LE = '<'

NATIVE = '='

4.6 bpack.np module

Numpy based codec for binary data structures.

class bpack.np.Codec(*descriptor*)

Bases: *Codec*

Numpy based codec.

(Unicode) strings are treated as "utf-8" encoded byte strings. UCS4 encoded strings are not supported.

decode(*data: bytes*, *count: int = 1*)

Decode binary data and return a record object.

encode(*record*)

Encode record (Python object) into binary data.

baseunits: *EBaseUnits* = 'bytes'

property dtype

Return the numpy *dtype* corresponding to the *codec.descriptor*.

bpack.np.**Decoder**

alias of *Codec*

class bpack.np.**ESignMode**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: `IntEnum`

Enumeration for sign encoding convention.

SIGNED = 1

SIGN_AND_MOD = 2

UNSIGNED = 0

bpack.np.**Encoder**

alias of *Codec*

bpack.np.**codec**(*cls*)

Class decorator to add (de)coding methods to a descriptor class.

The decorator automatically generates a *Codec* object form the input descriptor class and attach to it methods for conversion form/to bytes.

bpack.np.**decoder**(*cls*)

Class decorator to add (de)coding methods to a descriptor class.

The decorator automatically generates a *Codec* object form the input descriptor class and attach to it methods for conversion form/to bytes.

bpack.np.**descriptor_to_dtype**(*descriptor*) → numpy.dtype

Convert the descriptor of a binary record into a `numpy.dtype`.

Please note that (unicode) strings are treated as "utf-8" encoded byte strings. UCS4 encoded strings are not supported.

Sequences (`typing.Sequence` and `typing.List`) are always converted into `numpy.ndarray`.

See also:

bpack.descriptors.descriptor().

`bpack.np.`**`encoder`**(*cls*)

Class decorator to add (de)coding methods to a descriptor class.

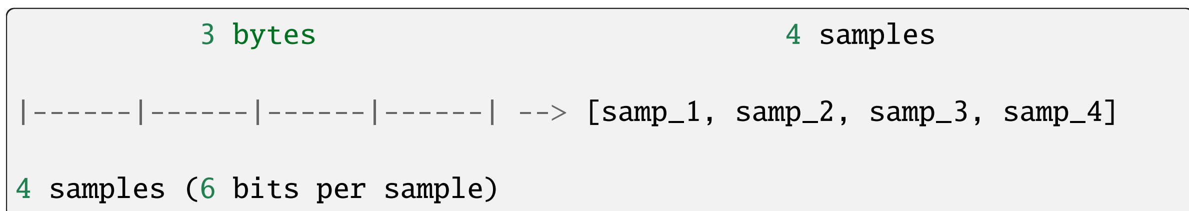
The decorator automatically generates a *Codec* object form the input descriptor class and attach to it methods for conversion form/to bytes.

`bpack.np.`**`unpackbits`**(*data: bytes*, *bits_per_sample: int*, *samples_per_block: int* | *None* = *None*, *bit_offset: int* = *0*, *blockstride: int* | *None* = *None*, *sign_mode:* ESignMode = *ESignMode.UNSIGNED*, *byteorder: str* = *'>'*, *use_lut: bool* = *True*) → numpy.ndarray

Unpack packed (integer) values form a string of bytes.

Takes in input a string of bytes in which (integer) samples have been stored using `bits_per_sample` bit for each sample, and returns the sequence of corresponding Python integers.

Example:



```
          3 bytes                             4 samples

|------|------|------|------| --> [samp_1, samp_2, samp_3, samp_4]

4 samples (6 bits per sample)
```

Parameters

- **`data`** – bytes string of bytes containing the packed data
- **`bits_per_sample`** – int the number of bits used to encode each sample
- **`samples_per_block`** – int, optional the number of samples in each data block contained in the input string of bytes. This parameter is mostly relevant if the data block contains other information (or padding bits) in addition to the data samples. The number of blocks is deduced from the length of the input string of bytes, the number of samples per block and the number of bits per sample. If *samples_per_block* is not provided it is assumed a single block, and the number of samples is derived from the length of the input string of bytes and the number of bits per sample.
- **`bit_offset`** – int, optional the number of bits after which the sequence of samples (data blocks) starts (default: 0). It can be used e.g. to take into account of a possible binary header at the beginning of the sequence of samples.

- **blockstride** – int, optional the number of bits between the start of a data block and the start of the following one. This parameter is mostly relevant if the data block contains other information (or padding bits) in addition to the data samples. If not provided the *blockstride* is assumed to be equal to the size of the data block i.e. *bits_per_sample * samples_per_block*.
- **sign_mode** – ESignMode, optional specifies how the sign of the integer samples shall is encoded. Dy default unsigned samples are assumed. .. seealso:: *ESignMode*.
- **byteorder** – str, optional Byte order of the encoded integers. Only relevant for multi byte samples. Default: ">" (big endian).
- **use_lut** – bool, optional specifies whenever the decoding of signed samples shall exploit look-up tables (typically faster). Default: True.

4.7 bpack.st module

Struct based codec for binary data structures.

class `bpack.st.`**`Codec`**(*descriptor*, *codec=None*, *decode_converters=None*, *encode_converters=None*)

Bases: `BaseStructCodec`

Struct based codec.

Default byte-order: MSB.

baseunits: *EBaseUnits* = **'bytes'**

`bpack.st.`**`Decoder`**

alias of *Codec*

`bpack.st.`**`Encoder`**

alias of *Codec*

`bpack.st.`**`codec`**(*cls*)

Class decorator to add (de)coding methods to a descriptor class.

The decorator automatically generates a *Codec* object form the input descriptor class and attach to it methods for conversion form/to bytes.

`bpack.st.`**`decoder`**(*cls*)

Class decorator to add (de)coding methods to a descriptor class.

The decorator automatically generates a *Codec* object form the input descriptor class and attach to it methods for conversion form/to bytes.

`bpack.st.encoder(cls)`

Class decorator to add (de)coding methods to a descriptor class.

The decorator automatically generates a *Codec* object form the input descriptor class and attach to it methods for conversion form/to bytes.

4.8 bpack.typing module

bpack support for type annotations.

`class bpack.typing.T(*args, **kwargs)`

Bases: `object`

Allow to specify numeric type annotations using string descriptors.

Example:

```
>>> T['u4']
typing.Annotated[int, TypeParams(byteorder=None,
                     type='int', size=4, signed=False)]
```

The resulting type annotation is a `typing.Annotated` numeric type with attached a *`bpack.typing.TypeParams`* instance.

String descriptors, or *typestr*, are compatible with numpy (a sub-set of one used in the numpy "array interface").

The *typestr* string format consists of 3 parts:

- an (optional) character describing the byte order of the data
  - <: little-endian,
  - >: big-endian,
  - `|`: not-relevant
- a character code giving the basic type of the array, and
- an integer providing the number of bytes the type uses

The basic type character codes are:

- `i`: sighed integer
- u: unsigned integer
- `f`: float
- `c`: complex
- S: bytes (string)

Note: *typestr* the format described above is a sub-set of the one used in the numpy "array interface".

See also:

`str_to_type_params()`, *TypeParams*, https://numpy.org/doc/stable/reference/arrays.dtypes.html and https://numpy.org/doc/stable/reference/arrays.interface.html

class bpack.typing.**TypeParams**(*byteorder: EByteOrder | None, type: Type[bool | int | float | complex | bytes | str], size: int | None, signed: bool | None*)

Bases: `NamedTuple`

Named tuple describing type parameters.

byteorder: *EByteOrder* **| None**

Alias for field number 0

signed: bool | None

Alias for field number 3

size: int | None

Alias for field number 2

type: Type[bool | int | float | complex | bytes | str]

Alias for field number 1

bpack.typing.**is_annotated**(*type_: Type*) → bool

Return True if the input is an annotated numeric type.

An *annotated numeric type* is assumed to be a `typing.Annotated` type annotation of a basic numeric type with attached a *bpack.typing.TypeParams* instance.

See also:

bpack.typing.T.

4.9 bpack.utils module

Utility functions and classes.

bpack.utils.**classdecorator**(*func*)

Class decorator that can be used with or without parameters.

bpack.utils.**create_fn**(*name, args, body, *, globals=None, locals=None, return_type=<dataclasses._MISSING_TYPE object>*)

Create a function object.

`bpack.utils.effective_type`(*type_: Type | Type[Enum], keep_annotations: bool = False*) → Type

Return the effective type.

In case of enums or sequences return the item type.

`bpack.utils.enum_item_type`(*enum_cls: Type[Enum]*) → Type

Return the type of the items of an enum.Enum.

This function also checks that all items of an enum have the same (or compatible) type.

`bpack.utils.is_enum_type`(*type_: Type*) → bool

Return True if the input is and `enum.Enum`.

`bpack.utils.is_int_type`(*type_: Type*) → bool

Return true if the effective type is an integer.

`bpack.utils.is_sequence_type`(*type_: Type, error: bool = False*) → bool

Return True if the input is an homogeneous typed sequence.

Please note that fields annotated with `typing.Tuple` are not considered homogeneous sequences even if all items are specified to have the same type.

`bpack.utils.sequence_type`(*type_: Type, error: bool = False*) → Type | None

Return the sequence type associated to a typed sequence.

The function return `list` or `tuple` if the input is considered a valid typed sequence, `None` otherwise.

Please note that fields annotated with `typing.Tuple` are not considered homogeneous sequences even if all items are specified to have the same type.

`bpack.utils.set_new_attribute`(*cls, name, value*)

Programmatically add a new attribute/method to a class.

CHAPTER

FIVE

DEVELOPERS GUIDE

5.1 Project links

PyPI page
: https://pypi.org/project/bpack

repository
: https://github.com/avalentino/bpack

issue tracker
: https://github.com/avalentino/bpack/issues

CI
: https://github.com/avalentino/bpack/actions

HTML documentation
: https://bpack.readthedocs.io

5.2 Set-up the development environment

5.2.1 Pip based environment

```
$ python3 -m venv --prompt venv .venv
$ source .venv/bin/activate
(venv) $ python3 -m pip install -r requirements-dev-txt
```

5.2.2 Conda based environment

```
$ conda create -c conda-forge -n bpack \
    --file requirements-dev.txt python=3
```

5.2.3 Debian/Ubuntu

```
$ sudo apt install python3-bitstruct python3-bitarray \
    python3-pytest python3-pytest-cov \
    python3-sphinx python3-sphinx-rtd-theme
```

5.3 Testing the code

5.3.1 Basic testing

```
$ python3 -m pytest
```

It is also recommended to use the `-W=error` option.

5.3.2 Advanced testing

Tox (>4) is used to run a comprehensive test suite on multiple Python version. It also checks formatting, coverage and ensures that the documentation builds properly.

```
$ tox run
```

5.4 Test coverage

```
$ python3 -m pytest --cov --cov-report=html --cov-report=term bpack
```

5.5 Check code style and formatting

The code style and formatting shall be checked with flake8 as follows:

```
$ python3 -m flake8 --statistics --count bpack
```

Moreover, also the correct formatting of "docstrings" shall be checked, using pydocstyle this time:

```
$ python3 -m pydocstyle --count bpack
```

A more strict check of formatting can be done using black:

```
$ python3 -m black --check bpack
```

Finally the ordering of imports can be checked with isort as follows:

```
$ python3 -m isort --check bpack
```

Please note that all the relevant configuration for the above mentioned tools are in the *pyproject.toml* file.

5.6 Build the documentation

```
$ make -C docs html
```

5.7 Test code snippets in the documentation

```
$ make -C docs doctest
```

5.8 Check documentation links

```
$ make -C docs linkcheck
```

5.9 Check documentation spelling

```
$ make -C docs spelling
```

5.10 Update the API documentation

```
$ rm -rf docs/api
$ sphinx-apidoc --module-first --separate --no-toc \
    --doc-project "bpack API" -o docs/api \
    --templatedir docs/_templates/apidoc \
    bpack bpack/tests
```

CHAPTER

SIX

COPYRIGHT AND LICENSE

Copyright 2020-2023 Antonio Valentino <antonio.valentino@tiscali.it>

Licensed under the Apache License, Version 2.0 (the “License”); you may not use this file except in compliance with the License. You may obtain a copy of the License at

> http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

6.1 Integral license text

```
                                 Apache License
                           Version 2.0, January 2004
                        http://www.apache.org/licenses/

   TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

   1. Definitions.

      "License" shall mean the terms and conditions for use, reproduction,
      and distribution as defined by Sections 1 through 9 of this␣
↪document.

      "Licensor" shall mean the copyright owner or entity authorized by
      the copyright owner that is granting the License.

      "Legal Entity" shall mean the union of the acting entity and all
      other entities that control, are controlled by, or are under common
```

(continues on next page)

(continued from previous page)

```
      control with that entity. For the purposes of this definition,
      "control" means (i) the power, direct or indirect, to cause the
      direction or management of such entity, whether by contract or
      otherwise, or (ii) ownership of fifty percent (50%) or more of the
      outstanding shares, or (iii) beneficial ownership of such entity.

      "You" (or "Your") shall mean an individual or Legal Entity
      exercising permissions granted by this License.

      "Source" form shall mean the preferred form for making␣
↪modifications,
      including but not limited to software source code, documentation
      source, and configuration files.

      "Object" form shall mean any form resulting from mechanical
      transformation or translation of a Source form, including but
      not limited to compiled object code, generated documentation,
      and conversions to other media types.

      "Work" shall mean the work of authorship, whether in Source or
      Object form, made available under the License, as indicated by a
      copyright notice that is included in or attached to the work
      (an example is provided in the Appendix below).

      "Derivative Works" shall mean any work, whether in Source or Object
      form, that is based on (or derived from) the Work and for which the
      editorial revisions, annotations, elaborations, or other␣
↪modifications
      represent, as a whole, an original work of authorship. For the␣
↪purposes
      of this License, Derivative Works shall not include works that␣
↪remain
      separable from, or merely link (or bind by name) to the interfaces␣
↪of,
      the Work and Derivative Works thereof.

      "Contribution" shall mean any work of authorship, including
      the original version of the Work and any modifications or additions
      to that Work or Derivative Works thereof, that is intentionally
      submitted to Licensor for inclusion in the Work by the copyright␣
↪owner
      or by an individual or Legal Entity authorized to submit on behalf␣
↪of
```

(continues on next page)

(continued from previous page)

```
      the copyright owner. For the purposes of this definition, "submitted
↪"
      means any form of electronic, verbal, or written communication sent
      to the Licensor or its representatives, including but not limited to
      communication on electronic mailing lists, source code control␣
↪systems,
      and issue tracking systems that are managed by, or on behalf of, the
      Licensor for the purpose of discussing and improving the Work, but
      excluding communication that is conspicuously marked or otherwise
      designated in writing by the copyright owner as "Not a Contribution.
↪"

      "Contributor" shall mean Licensor and any individual or Legal Entity
      on behalf of whom a Contribution has been received by Licensor and
      subsequently incorporated within the Work.

   2. Grant of Copyright License. Subject to the terms and conditions of
      this License, each Contributor hereby grants to You a perpetual,
      worldwide, non-exclusive, no-charge, royalty-free, irrevocable
      copyright license to reproduce, prepare Derivative Works of,
      publicly display, publicly perform, sublicense, and distribute the
      Work and such Derivative Works in Source or Object form.

   3. Grant of Patent License. Subject to the terms and conditions of
      this License, each Contributor hereby grants to You a perpetual,
      worldwide, non-exclusive, no-charge, royalty-free, irrevocable
      (except as stated in this section) patent license to make, have␣
↪made,
      use, offer to sell, sell, import, and otherwise transfer the Work,
      where such license applies only to those patent claims licensable
      by such Contributor that are necessarily infringed by their
      Contribution(s) alone or by combination of their Contribution(s)
      with the Work to which such Contribution(s) was submitted. If You
      institute patent litigation against any entity (including a
      cross-claim or counterclaim in a lawsuit) alleging that the Work
      or a Contribution incorporated within the Work constitutes direct
      or contributory patent infringement, then any patent licenses
      granted to You under this License for that Work shall terminate
      as of the date such litigation is filed.

   4. Redistribution. You may reproduce and distribute copies of the
      Work or Derivative Works thereof in any medium, with or without
      modifications, and in Source or Object form, provided that You
```

(continues on next page)

(continued from previous page)

```
   meet the following conditions:

   (a) You must give any other recipients of the Work or
       Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
       stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works
       that You distribute, all copyright, patent, trademark, and
       attribution notices from the Source form of the Work,
       excluding those notices that do not pertain to any part of
       the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its
       distribution, then any Derivative Works that You distribute must
       include a readable copy of the attribution notices contained
       within such NOTICE file, excluding those notices that do not
       pertain to any part of the Derivative Works, in at least one
       of the following places: within a NOTICE text file distributed
       as part of the Derivative Works; within the Source form or
       documentation, if provided along with the Derivative Works; or,
       within a display generated by the Derivative Works, if and
       wherever such third-party notices normally appear. The contents
       of the NOTICE file are for informational purposes only and
       do not modify the License. You may add Your own attribution
       notices within Derivative Works that You distribute, alongside
       or as an addendum to the NOTICE text from the Work, provided
       that such additional attribution notices cannot be construed
       as modifying the License.

   You may add Your own copyright statement to Your modifications and
   may provide additional or different license terms and conditions
   for use, reproduction, or distribution of Your modifications, or
   for any such Derivative Works as a whole, provided Your use,
   reproduction, and distribution of the Work otherwise complies with
   the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
   any Contribution intentionally submitted for inclusion in the Work
   by You to the Licensor shall be under the terms and conditions of
   this License, without any additional terms or conditions.
   Notwithstanding the above, nothing herein shall supersede or modify
```

(continues on next page)

(continued from previous page)

```
      the terms of any separate license agreement you may have executed
      with Licensor regarding such Contributions.

   6. Trademarks. This License does not grant permission to use the trade
      names, trademarks, service marks, or product names of the Licensor,
      except as required for reasonable and customary use in describing␣
↪the
      origin of the Work and reproducing the content of the NOTICE file.

   7. Disclaimer of Warranty. Unless required by applicable law or
      agreed to in writing, Licensor provides the Work (and each
      Contributor provides its Contributions) on an "AS IS" BASIS,
      WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
      implied, including, without limitation, any warranties or conditions
      of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
      PARTICULAR PURPOSE. You are solely responsible for determining the
      appropriateness of using or redistributing the Work and assume any
      risks associated with Your exercise of permissions under this␣
↪License.

   8. Limitation of Liability. In no event and under no legal theory,
      whether in tort (including negligence), contract, or otherwise,
      unless required by applicable law (such as deliberate and grossly
      negligent acts) or agreed to in writing, shall any Contributor be
      liable to You for damages, including any direct, indirect, special,
      incidental, or consequential damages of any character arising as a
      result of this License or out of the use or inability to use the
      Work (including but not limited to damages for loss of goodwill,
      work stoppage, computer failure or malfunction, or any and all
      other commercial damages or losses), even if such Contributor
      has been advised of the possibility of such damages.

   9. Accepting Warranty or Additional Liability. While redistributing
      the Work or Derivative Works thereof, You may choose to offer,
      and charge a fee for, acceptance of support, warranty, indemnity,
      or other liability obligations and/or rights consistent with this
      License. However, in accepting such obligations, You may act only
      on Your own behalf and on Your sole responsibility, not on behalf
      of any other Contributor, and only if You agree to indemnify,
      defend, and hold each Contributor harmless for any liability
      incurred by, or claims asserted against, such Contributor by reason
      of your accepting any such warranty or additional liability.
```

(continues on next page)

(continued from previous page)


```
END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

   To apply the Apache License to your work, attach the following
   boilerplate notice, with the fields enclosed by brackets "[]"
   replaced with your own identifying information. (Don't include
   the brackets!)  The text should be enclosed in the appropriate
   comment syntax for the file format. We also recommend that a
   file or class name and description of purpose be included on the
   same "printed page" as the copyright notice for easier
   identification within third-party archives.

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or␣
↪implied.
See the License for the specific language governing permissions and
limitations under the License.
```

CHAPTER

SEVEN

RELEASE NOTES

7.1 bpack v1.2.0 (UNRELEASED)

- Drop support to Python 3.7 and 3.8. Now *bpack* requires Python >= 3.9.
- New 'full' installation option added to *pyproject.toml*.
- No longer use deprecated syntax in sphinx configuration.
- Improved documentation and fixed typos.
- flake8 configuration moved to a dedicated file.
- Do not test the `bpack.ba` backend in PyPy3.
- New functions: - `bpack.typing.type_params_to_str()` - `bpack.descriptors.flat_fields_iterator()`
- New `bpack.tools.codegen` module. It includes tool to generate flat binary record descriptors starting from nested ones (requires Python >= 3.10).

7.2 bpack v1.1.0 (15/04/2023)

- Added support for signed integers to `bpack.np.unpackbits()`. Both standard signed integers and integers encoded with sign and module are now supported.
- Use uppercase enums in *s1isp.py* example.
- Improved docstrings in `bpack.np`.
- Fixed several typos.

7.3 bpack v1.0.0 (05/02/2023)

- Fix compatibility with Python v3.11.
- Move setup configuration from *setup.cfg* to *pyproject.toml*.

7.4 bpack v0.8.2 (20/03/2022)

- Fallback to standard bitstruct if the bitstruct.c extension does not support the format string

7.5 bpack v0.8.1 (30/11/2021)

- Drop `setup.py`, no longer needed.
- Improve compatibility with typing-extensions v4.0 (closes gh-1).
- Use the compiled extension of bitstruct when available (and compatible with the specified format string).
- Use cbitsturct when available (preferred over the compiled extension of bitstruct).

7.6 bpack v0.8.0 (03/06/2021)

- New "encoding" feature. Records can be now encoded into binary strings using the `bpack.st` and `bpack.bs` backends. Previously only "decoding" was supported. The `bpack.np` only implements a partial support to encoding currently.

7.7 bpack v0.7.1 (08/03/2021)

- Improved User Guide
- `bpack.np.unpackbits()` has been generalized and optimized.
- New example for packet decoding.
- Improved support for nested records.

7.8 bpack v0.7.0 (21/01/2021)

- New *packbit/unpackbit* functions (provisional API).
- Fixed a bug in decoding of nested records.
- Added example program for Sentinel-1 space packets decoding

7.9 bpack v0.6.0 (15/01/2021)

- New numpy based backend.
- New `bpack.enums.EByteOrder.get_native()` method.
- Now data types in descriptor definition can also be specified by means of special type annotation type (`bpack.typing.T`) that accepts numpy-like format strings.
- Now it is no longer necessary to use the `dataclasses.dataclass()` decorator to define a descriptor. That way to define descriptors is **depercated**. All parameters previously specified via `dataclasses.dataclass()` (like e.g. *frozen*) shall now be passed directly to the `bpack.descriptors.descriptor()` decorator. With this change the use of `dataclasses` becomes an implementation detail.
- The `size` parameter of the `bpack.descriptors.field()` factory function is now optional.
- General improvements and code refactoring.
- Improved CI testing.
- Added automatic spell checking of documentation in CI.
- Backward incompatible changes:
  - `bpack.enums.EBaseUnits`, `bpack.enums.EByteOrder` and `bpack.enums.EBitOrder` enums moved to the new `bpack.enums` module (the recommended way to access enums is directly from `bpack`, e.g. `bpack.EByteOrder`)
  - `bpack.enums.EByteOrder.BIG` and `bpack.enums.EByteOrder.LITTLE` enumerates have been renamed into `bpack.enums.EByteOrder.BE` and `bpack.enums.EByteOrder.LE` respectively
  - classes decorated with the `bpack.descriptors.descriptor()` decorator no longer have the `__len__` method automatically added; the recommended way to compute the size of a descriptors (class or instance) is to use the `bpack.descriptros.calcsize()` function
  - the default behavior of the `bpack.decorators.calcsize()` has been changed to return the size of the input *descriptor* in the same *base units* of the descriptor itself; previously the default behavior was to return the size in bytes

7.10 bpack v0.5.0 (31/12/2020)

- Initial release.

  The package implements all core functionalities but

  - the API is still not stable
  - the documentation is incomplete
  - some advanced feature is still missing

INDEX

A

asdict() (*in module bpack.descriptors*), 29
astuple() (*in module bpack.descriptors*), 30

B

baseunits (*bpack.ba.Decoder attribute*), 25
baseunits (*bpack.bs.Codec attribute*), 26
baseunits (*bpack.np.Codec attribute*), 34
baseunits (*bpack.st.Codec attribute*), 36
baseunits() (*in module bpack.descriptors*), 30
BE (*bpack.enums.EByteOrder attribute*), 33
BinFieldDescriptor (*class in bpack.descriptors*), 28
bitorder() (*in module bpack.descriptors*), 30
BITS (*bpack.enums.EBaseUnits attribute*), 32
bpack
    module, 25
bpack.ba
    module, 25
bpack.bs
    module, 26
bpack.codecs
    module, 27
bpack.descriptors
    module, 28
bpack.enums
    module, 32
bpack.np
    module, 33
bpack.st
    module, 36
bpack.typing
    module, 37
bpack.utils
    module, 38
byteorder (*bpack.typing.TypeParams attribute*), 38
byteorder() (*in module bpack.descriptors*), 30
BYTES (*bpack.enums.EBaseUnits attribute*), 32

C

calcsize() (*in module bpack.descriptors*), 30
classdecorator() (*in module bpack.utils*), 38
Codec (*class in bpack.bs*), 26
Codec (*class in bpack.codecs*), 27
Codec (*class in bpack.np*), 33
Codec (*class in bpack.st*), 36
codec() (*in module bpack.bs*), 26
codec() (*in module bpack.np*), 34
codec() (*in module bpack.st*), 36
compare (*bpack.descriptors.Field attribute*), 29
create_fn() (*in module bpack.utils*), 38

D

decode() (*bpack.ba.Decoder method*), 25
decode() (*bpack.codecs.Decoder method*), 27
decode() (*bpack.np.Codec method*), 33
Decoder (*class in bpack.ba*), 25
Decoder (*class in bpack.codecs*), 27
Decoder (*in module bpack.bs*), 26
Decoder (*in module bpack.np*), 34
Decoder (*in module bpack.st*), 36
decoder() (*in module bpack.ba*), 25
decoder() (*in module bpack.bs*), 26
decoder() (*in module bpack.np*), 34
decoder() (*in module bpack.st*), 36
default (*bpack.descriptors.Field attribute*), 29

DEFAULT (*bpack.enums.EBitOrder attribute*), 33
DEFAULT (*bpack.enums.EByteOrder attribute*), 33
default_factory (*bpack.descriptors.Field attribute*), 29
descriptor() (*in module bpack.descriptors*), 30
descriptor_to_dtype() (*in module bpack.np*), 34
dtype (*bpack.np.Codec property*), 34

E

EBaseUnits (*class in bpack.enums*), 32
EBitOrder (*class in bpack.enums*), 32
EByteOrder (*class in bpack.enums*), 33
effective_type() (*in module bpack.utils*), 39
encode() (*bpack.codecs.Encoder method*), 28
encode() (*bpack.np.Codec method*), 34
Encoder (*class in bpack.codecs*), 27
Encoder (*in module bpack.bs*), 26
Encoder (*in module bpack.np*), 34
Encoder (*in module bpack.st*), 36
encoder() (*in module bpack.bs*), 26
encoder() (*in module bpack.np*), 35
encoder() (*in module bpack.st*), 36
enum_item_type() (*in module bpack.utils*), 39
ESignMode (*class in bpack.np*), 34

F

Field (*class in bpack.descriptors*), 29
field() (*in module bpack.descriptors*), 31
field_descriptors() (*in module bpack.descriptors*), 31
fields() (*in module bpack.descriptors*), 31
flat_fields_iterator() (*in module bpack.descriptors*), 31

G

get_field_descriptor() (*in module bpack.descriptors*), 32
get_native() (*bpack.enums.EByteOrder class method*), 33

H

has_codec() (*in module bpack.codecs*), 28
hash (*bpack.descriptors.Field attribute*), 29

I

init (*bpack.descriptors.Field attribute*), 29
is_annotated() (*in module bpack.typing*), 38
is_descriptor() (*in module bpack.descriptors*), 32
is_enum_type() (*bpack.descriptors.BinFieldDescriptor method*), 28
is_enum_type() (*in module bpack.utils*), 39
is_field() (*in module bpack.descriptors*), 32
is_int_type() (*bpack.descriptors.BinFieldDescriptor method*), 28
is_int_type() (*in module bpack.utils*), 39
is_sequence_type() (*bpack.descriptors.BinFieldDescriptor method*), 28
is_sequence_type() (*in module bpack.utils*), 39

K

kw_only (*bpack.descriptors.Field attribute*), 29

L

LE (*bpack.enums.EByteOrder attribute*), 33
LSB (*bpack.enums.EBitOrder attribute*), 33

M

metadata (*bpack.descriptors.Field attribute*), 29
module
  bpack, 25
  bpack.ba, 25
  bpack.bs, 26
  bpack.codecs, 27
  bpack.descriptors, 28
  bpack.enums, 32
  bpack.np, 33
  bpack.st, 36
  bpack.typing, 37
  bpack.utils, 38

MSB (*bpack.enums.EBitOrder attribute*), 33

N

name (*bpack.descriptors.Field attribute*), 29
NATIVE (*bpack.enums.EByteOrder attribute*), 33

O

offset (*bpack.descriptors.BinFieldDescriptor attribute*), 29

P

packbits() (*in module bpack.bs*), 26

R

repeat (*bpack.descriptors.BinFieldDescriptor attribute*), 29
repr (*bpack.descriptors.Field attribute*), 29

S

sequence_type() (*in module bpack.utils*), 39
set_field_descriptor() (*in module bpack.descriptors*), 32
set_new_attribute() (*in module bpack.utils*), 39
SIGN_AND_MOD (*bpack.np.ESignMode attribute*), 34
signed (*bpack.descriptors.BinFieldDescriptor attribute*), 29
SIGNED (*bpack.np.ESignMode attribute*), 34
signed (*bpack.typing.TypeParams attribute*), 38
size (*bpack.descriptors.BinFieldDescriptor attribute*), 29
size (*bpack.typing.TypeParams attribute*), 38

T

T (*class in bpack.typing*), 37
total_size (*bpack.descriptors.BinFieldDescriptor property*), 29
type (*bpack.descriptors.BinFieldDescriptor attribute*), 29
type (*bpack.descriptors.Field attribute*), 29
type (*bpack.typing.TypeParams attribute*), 38
TypeParams (*class in bpack.typing*), 38

U

unpackbits() (*in module bpack.bs*), 27
unpackbits() (*in module bpack.np*), 35
UNSIGNED (*bpack.np.ESignMode attribute*), 34
update_from_type() (*bpack.descriptors.BinFieldDescriptor method*), 29

V

validate() (*bpack.descriptors.BinFieldDescriptor method*), 29