Destructify Documentation

Release 0.2.0

Ralph Broenink

Mar 23, 2019

Contents

Destructify is a Pythonic and pure-Python 3 method to express binary data, allowing you to read and write binary structures. You simply specify a structure by creating a class as follows:

```
class ExampleStructure(destructify.Structure):
    some_number = destructify.IntegerField(default=0x13, length=4, byte_order='little
↪', signed=True)
    length = destructify.IntegerField(length=1)
    data = destructify.FixedLengthField(length='length')
```

Now you can parse your own binary data:

```
example = ExampleStructure.from_bytes(b"\x01\x02\x03\x04\x0BHello world")
print(example.data)  # b'Hello world'
```

Or write your own data:

```
example2 = ExampleStructure(data=b'How are you doing?')
print(bytes(example2))  # b'\x13\x00\x00\x00\x12How are you doing?'
```

Contents:

CHAPTER 1

Structures

Destructify uses structures to define how to parse binary data structures. If you have used Django before, you may see some resemblance with how models are defined in that project. Don't worry if you don't know anything about Django, as the following is everything you need to know:

- Each structure is a Python class that subclasses *Structure*
- Each attribute of the structure defines a field in the binary structure

All of this allows you to write a very clean looking specification of binary data structures that is easy to write, but also trivial to read and comprehend. Some of this even resembles parts of C-style structures, so it can be dead simple to write some code to interface between C programs and Python programs.

1.1 Simple example

Let's say we have some simple C-style structure that allows you to write your name (in a fixed-length fashion), your birth year and your balance with some company (ignoring the cents). This might look like the following in C:

```
struct {
    char name[24];
    uint16_t birth_year;
    int32_t balance;
} Person;
```

In Destructify, you would specify this as follows:

```
import destructify

class Person(destructify.Structure):
    name = destructify.StringField(length=5, encoding='utf-8')
    birth_year = destructify.IntegerField(length=2, signed=False)
    balance = destructify.IntegerField(length=4, signed=True)
```

(continues on next page)

(continued from previous page)

```
    class Meta:
        byte_order = 'big'
```

Each of the attributes above are called fields. Each field is specified as a class attribute, and each attribute defines how it parses this part of the structure. Also note that ordering matters, and fields are parsed in the order they are defined in.

You may also have noticed that we have defined a `Meta` inner class containing the `Meta.byte_order` attribute. This is required for the two *`IntegerField`* we use. When writing binary data, the byte order, or endianness as it is also commonly called, specifies how bytes are read and written. You can specify this as a default on a per-structure basis or specifically on a per-field basis.

You can now start using this structure. Reading a structure is as easy as calling the class-method *`Structure.from_bytes()`* as follows:

```
>>> person = Person.from_bytes(b"Bobby\x07\xda\x00\x00\x00\xc8")
<Person: Person(name='Bobby', birth_year=2010, balance=200)>
```

From the resulting object, you can simply access the different attributes:

```
>>> person.name
Bobby
>>> person.birth_year
2010
```

Creating a structure is also very simple, as you can pass all attributes to the constructor of the structure, or change their value as attribute. Obtaining the binary structure is then as easy as converting the object to `bytes`:

```
>>> Person(name="Carly", birth_year=1993, balance=-100)
>>> person.name = "Alice"
>>> bytes(person)
b"Alice\x07\xc9\xff\xff\xff\x9c"
```

1.2 C-style operations

Continuing our above example of a C-style struct, we know that we can also obtain the size of a structure in C using the `sizeof` function. We can do the same in Destructify using `len`:

```
>>> len(Person)
11
```

This is only possible when we use fixed-length fields. If we have some field somewhere that is of variable length, we can't determine this length anymore:

```
>>> class FlexibleStructure(destructify.Structure):
...     field = destructify.StringField(terminator=b'\0')
...
>>> len(FlexibleStructure)
Traceback (most recent call last):
    (...)
destructify.exceptions.ImpossibleToCalculateLengthError
```

Similarly, you can use *`Structure.as_cstruct()`* to see how you'd write the same structure in a C-style struct. Note that

1.3 Field types

In the first example, we have shown some field types, but Destructify comes with dozens of different built-in fields. Each of these is used to define how a piece of bytes is to be interpreted and how it is to be written to bytes again.

It is not possible to make a general assumption about all fields, but most fields combine different methods of consuming and writing data to and from a stream, with a single Python representation. Taking the `StringField` as an example, you may have noticed that we are only able to fit 5-byte names in this field. What if we had longer or shorter names? Luckily, `StringField` allows you to pass different keyword-arguments to define how this works.

Reading through *Built-in fields specification* you will discover that all fields have a smorgasbord of different attributes to control how they read, convert and parse values to and from a stream. To illustrate what we mean, we show you how `BytesField` has different operating modes in the next section.

But remember, you can always implement your own field if none of the built-in fields does what you want.

1.4 Controlling a field through attributes

Most fields take the `BytesField` as a base class, as this field has various common options for parsing bytes from a stream. Two of the most common cases, a fixed-length field, and a field 'until' some byte sequence, are possible. It is even possible to make this a lot more complex, as we try to show in five examples:

`BytesField(length=5)` This reads exactly the specified amount of bytes from the field, and returns that immediately.

`BytesField(length=20, padding=b' ')` This is a variant of the previous example, that allows for some variance in the field: 20 bytes are read and all spaces are removed from right-to-left. When writing, spaces are automatically added as well.

`BytesField(terminator=b'\0')` This form allows us to read until a single NULL-byte is encountered. This is typically how strings are represented in C, and are called NULL-terminated strings. The advantage of this is that the value can take any length, as long as it is terminated with a NULL-byte (and the value itself does not contain any NULL-bytes).

Using this has some disadvantages, as it is not possible to use `Field.lazy` on such a field: it must be parsed in its entirety to know its length.

`BytesField(length=20, terminator=b'\0')` This form combines the two methods by specifying both a fixed amount of bytes, *and* a terminator. This is a common model when writing strings to fixed-length buffers in C: it reads 20 bytes from the stream, and then looks for the terminator.

This is different from specifying a length with padding, as this allows junk to exist in the padding of the field. That may occur commonly in C: imagine you declare a buffer of fixed length, but do not properly fill it with zeroes. In that case, some random bytes may exist in the padding, not just NULL-bytes.

Note that this field does not know how to write a value that is too short, as padding has nog been defined yet; but there is a solution:

`BytesField(length=20, terminator=b'\0', padding=b'\0')` This is the best of all worlds, allowing us to read 20 bytes, terminate the relevant part at the NULL-terminator while reading, and allow us to write shorter-length values as these will be padded with NULL-bytes. This is usually how you'd implement fixed-length C-style strings.

As you can see from these five examples, it highly depends on how your structure looks like what you'd define in the structure. Again, these are only examples, and you should read *Built-in fields specification* to get an idea of all of the options for all of the built-in fields.

1.5 Streams

Until now, you may have noticed we have been using *Structure.from_bytes()* and *Structure.to_bytes()* to convert from and to bytes. In fact, these are convenience methods, as Destructify actually works on streams. You can use this to simply open a file and parse this, without needing to convert it to bytes first:

```
with open("file.png", "rb") as f:
    structure = MyStructure.from_stream(f)
```

This allows you to read in large files into a Python structure.

1.6 Structure methods

Apart from the way we define the fields in a structure, all structures are normal Python classes and can add additional functions and calculated properties. This is helpful, as you can use this to create per-instance methods that allow you to work on a particular instance of your structure, and keep your business logic in one place:

```
class Person(destructify.Structure):
   name = destructify.StringField(length=5, encoding='utf-8')
   birth_year = destructify.IntegerField(length=2, signed=False)
   balance = destructify.IntegerField(length=4, signed=True)

   class Meta:
       byte_order = 'big'

   def add_to_balance(self, amount):
       """Adds the given amount to the balance of this person."""
       self.balance += amount

   @property
   def age(self):
       """The most naive method of determining the age of the person."""
       import datetime
       return datetime.date.today().year - self.birth_year
```

Note that we have implemented the last method in this example as a property, showing how you would implement a calculated property that is not written to the binary structure.

The *Structure* defines some function of its own, for instance the *Structure.to_stream()* method. You're free to override these functions to do whatever you like. An example would be:

```
class Person(destructify.Structure):
   ...

   def to_stream(self, *args, **kwargs):
       do_something()
       result = super().to_stream(*args, **kwargs)
       do_more()
       return result
```

In this example, we do something just before we write the data to a stream. It's important to call the superclass method if you want to retain original behaviour and return its value (that's what that `super()` call is for). Also note that we pass the original arguments of the function through to the original function, without defining what these are precisely.

As it is common to modify some fields just before they have been written, you may also choose to override *Structure.finalize*.

1.7 The Meta class

You may have noticed that we use a class named `Structure.Meta` in some of our definitions. You can use this class to specify some global attributes for your structure. For instance, this allows you to set some defaults on some fields, e.g. the *`StructureOptions.byte_order`*.

The Meta attributes you define, are available in the *`Structure._meta`* attribute of the structure. This is a `StructureOptions` object.

The following options are available:

`StructureOptions`**.structure_name**
: The name of the structure. Defaults to the class name of the structure.

`StructureOptions`**.byte_order**
: The default byte-order for fields in this structure. Is not set by default, and can be `little` or `big`.

`StructureOptions`**.encoding**
: The default character encoding for fields in this structure. Defaults to `utf-8`.

`StructureOptions`**.alignment**
: Can be set to a number to align the start of all fields. For instance, if this is `4`, the start of all fields will be aligned to 4-byte multiples; meaning that, after a 2-byte field, a 2-byte gap will automatically be added. This is useful for e.g. C-style structs, that are automatically aligned.

  This alignment does *not* apply when *`Field.offset`* or *`Field.skip`* is set. When using subsequent *`BitField`* s, this may also be ignored.

  See also:

  The Lost Art of Structure Packing Some background information about alignment of C-style structures.

`StructureOptions`**.checks**
: This is a list of checks to execute after parsing the *`Structure`*, or just before writing it. Every check must be a function that accepts a *`ParsingContext.f`* object, and return a truthy value when the check is successful. For instance:

```
class Struct(Structure):
    value = IntegerField(length=1)
    checksum = IntegerField(length=1)

    class Meta:
        checks = [
            lambda f: (f.value1 * 2 % 256) == f.checksum
        ]
```

  When any of the checks fails, a `CheckError` is raised.

`StructureOptions`**.capture_raw**
: If True, requests the *`ParsingContext`* to capture raw bytes for all fields in the structure.

CHAPTER 2

Advanced parsing

In the previous chapter, we have covered generally how you'd define a simple structure. However, there is much more ground to cover there, so we'll take a deeper dive into how parsing works in Destructify.

2.1 Depending on other fields

Until now, we have been using fixed length fields, without any dependency on other fields. However, it is not untypical for a field to have its length set by some other property. Take the following example:

```
import destructify

class DependingStructure(destructify.Structure):
    length = destructify.IntegerField(1)
    content = destructify.BytesField(length='length')
```

Since the *BytesField.length* attribute is special and allows you to set a string referencing another field, you can now simply do the following:

```
>>> DependingStructure(content=b"hello world").to_bytes()
b'\x0bhello world'
>>> DependingStructure.from_bytes(b'\x06hello!')
<DependingStructure: DependingStructure(length=6, content=b'hello!')>
```

Actually, there's some magic involved here, and that centers around the *ParsingContext* class. This class is passed around while parsing from and writing to a stream, and filled with information about the current process. This allows you to reference fields that have been parsed before the current field. This is what happens when you pass a string to the *BytesField.length* attribute: it is interpreted as a field name and obtained from the context while parsing and writing the data.

2.2 Calculating attributes

The *BytesField.length* attribute actually allows you to provide a callable as well. This callable takes a single argument, which is a *ParsingContext.f* object. This is a special object that allows you to transparently access other fields during parsing. This allows you to write more advanced calculations if you need to, or add multiple fields together:

```
class DoubleLengthStructure(destructify.Structure):
    length1 = destructify.IntegerField(1)  # multiples of 10 (for some reason)
    length2 = destructify.IntegerField(1)
    content = destructify.BytesField(length=lambda c: c.length1 * 10 + c.length2)
```

class destructify.**this**

As `lambda` functions can become quite tiresome to write out, it is also possible to use the special *this* object to write this. The *this* object is a higher-level lazily parsed object that constructs `lambda` functions for you. This is better shown by example, as these are equivalent:

```
this.field + this.field2 * 3
lambda this: this.field + this.field2 * 3
```

Writing the same structure again, we could also do the following:

```
import destructify
from destructify import this

class DoubleLengthStructure(destructify.Structure):
    length1 = destructify.IntegerField(1)
    length2 = destructify.IntegerField(1)
    content = destructify.BytesField(length=this.length1 * 10 + this.length2)
```

Note that this lazy object can do most normal arithmetic, but unfortunately, Python does not allow us to override the `len` function to return a lazy object. Therefore, you can use `len_` as a lazy alternative.

2.3 Overriding values

Having shown how we can read values without much problem, being able to write values is also quite important for structures. We know from previous examples that this works without much issues:

```
>>> DependingStructure(content=b"hello world").to_bytes()
b'\x0bhello world'
```

That begs the question: how does `length` know that it know that it needs to get the length from the `content` field? That is because there's something else going on in the background: when set to a string, the *BytesField* automatically specifies the *Field.override* of the `length` field to be set to another value, just before it is being written.

This is nice and all, but what if the length is actually some calculation that is more advanced than simply taking the length? For instance, what if the length field includes its own length? This is also very easy!

```
import destructify

class DependingStructure(destructify.Structure):
    length = destructify.IntegerField(length=4, byte_order='big', signed=False,
```

(continues on next page)

(continued from previous page)

```
                                    override=lambda c, v: len(c.content) + 4)
    content = destructify.BytesField(length=lambda c: c.length - 4)
```

As you can spot, we now explicitly state using lambda functions how to get the length when we are reading the field, and also how to set the length when we are writing the field.

As with the *BytesField.length* we defined before, the *Field.override* we have specified, receives a *ParsingContext.f*, but also the current value.

Several fields allow you to specify advanced structures such as these, allowing you to dynamically modify how your structure is built. See *Built-in fields specification* for a full listing of all the fields and how you can specify calculated values.

2.4 How a structure is read and written

We have now seen how *Field.override* works, but there are more ways to parse and write more advanced structures. You can alter the behaviour of a field by e.g. specifying *Field.decoder* and *Field.encoder*, or use functions on the *Structure* to modify values, while it is being parsed.

All these hooks can become quite complex, so the list below shows how a value is parsed from a stream into a *Structure* and vice versa.

The following functions are called on a value while reading from a stream by *Structure.from_stream()*:

- *Field.seek_start()* searches the start of the value in the stream, implementing e.g. *Field.skip*
- *Field.from_stream()* reads the value from the stream and adjusts it to a Python representation
- *Field.decode_value()* is called on the value retrieved from the stream to convert it to the proper Python value, implementing *Field.decoder*.
- *Field.get_initial_value()* is a function that is intended to adjust the value based on other fields, which is an empty hook function (at this point).
- *Structure.initialize()* is called to allow you for some final adjustments

If the field is *Field.lazy*, parsing goes a little bit differently, as *Field.from_stream()* and *Field.decode_value()* are delayed:

- *Field.seek_start()* searches the start of the value in the stream
- *Field.seek_end()* to seek the end of the value in the stream, but only if there's a next field with a relative offset
- *Field.get_initial_value()* is called, passing a Proxy object
- *Structure.initialize()* is called

And the following methods are called before writing to a stream by *Structure.to_stream()*:

- *Field.get_final_value()* is called on all values in the structure, implementing *Field.override*.
- *Structure.finalize()* is called to allow you to make some final adjustments
- *Field.encode_value()* is called on the value to convert it to a Python value that can be passed down, implementing *Field.encoder*.
- *Field.seek_start()* searches the start of the value in the stream, implementing e.g. *Field.skip*
- *Field.to_stream()* writes the value to the stream

Note that the two lists are intentionally not entirely symmetrical: individual field finalizers/initializers are in both cases called before the structure finalizer/initializer. Additionally, there's no equivalent for *Field.override* while reading the field, as that makes less sense. The hook is there, however.

In the chapters *Custom fields* and *Built-in fields specification*, we'll dive deeper into overriding these methods.

2.5 Decoding/encoding values

In some cases, you only may to modify a field a little bit. For instance, the value that is written to the stream is off-by-one, or you wish to return a value of a different type. As this is such a common use case, you can simply write a *Field.decoder*/*Field.encoder* pair for post-processing the value. It sits right between the parsing of the field, and the writing to the structure; from the perspective of the structure, this is how the field returned the value, whereas the field is unaware of something happening with the value.

Let's say that we are reading a date, but the value in the stream is in years since 2000, and the month is off-by-one in the stream. Then, we would write this:

```
class DateStructure(destructify.Structure):
    year = destructify.BitField(length=7, decoder=lambda v: v + 2000, encoder=lambda␣
↪v: v - 2000)
    month = destructify.BitField(length=4, decoder=lambda v: v + 1, encoder=lambda v:␣
↪v - 1)
    day = destructify.BitField(length=5)
```

You can even change the return type of the value. And since the callable for *Field.decoder* and *Field.encoder* takes a single argument, you can even simply do this:

```
import ipaddress

class IPStructure(destructify.Structure):
    ip = destructify.IntegerField(length=4, byte_order='big',
                                  decoder=ipaddress.IPv4Address, encoder=int)
```

While doing this, you can easily break the idempotency of a field (see *Custom fields*), so you are recommended to treat these attributes as a pair; although it is not required, allowing you to create some esoteric structures.

See *Custom fields* for how you can change the way a field works more significantly.

2.6 Offset, skip and alignment

It can happen that information in your structure is scattered throughout the stream. For instance, it can happen that a header specifies where to find the data in the stream. You can use *Field.offset* to specify an absolute offset in the stream, given an integer or a field value:

```
>>> class OffsetStructure(destructify.Structure):
...     offset = destructify.IntegerField(length=4, byte_order='big', signed=False)
...     length = destructify.IntegerField(length=4, byte_order='big', signed=False)
...     content = destructify.BytesField(offset='offset', length='length')
...
>>> OffsetStructure.from_bytes(b'\0\0\0\x10\0\0\0\x05paddingxhello')
<OffsetStructure: OffsetStructure(offset=16, length=5, content=b'hello')>
```

If you need to specify a offset from the end of the stream, a negative value is also possible. During writing, this is a little bit ambiguous, so you must be careful how you'd define this.

Remember that fields are always parsed in their defined order, and a field that follows a offset field, will continue parsing where the previous field left off.

If you need to skip a few bytes from the previous field, you can use *Field.skip*. You can use this to skip some padding without defining a field specifically to parse the padding. This is something that happens commonly when the stream is aligned to some multibyte offset, which can also be defined globally for the structure:

```
>>> class AlignedStructure(destructify.Structure):
...     field1 = destructify.IntegerField(length=1)
...     field2 = destructify.IntegerField(length=1)
...
...     class Meta:
...         alignment = 4
...
>>> AlignedStructure.from_bytes(b"\x01pad\x02pad")
<AlignedStructure: AlignedStructure(field1=1, field2=2)>
```

2.7 Lazily parsing fields

It can happen that you have a structure that reads huge chunks of data from the stream, but you don't want to keep all of this in memory while you are parsing from the stream. You can make fields lazy to defer their parsing to a later point in time.

To support this, Destructify uses a Proxy object, that is returned by the parser instead of the actual resulting value. This Proxy object can be used as you'd normally use the value, but it is only resolved from the stream as soon as it is actually required. For instance:

```
>>> class LazyStructure(destructify.Structure):
...     huge_content = destructify.BytesField(length=200, lazy=True)
...
>>> l = LazyStructure.from_bytes(b"a"*200)
>>> type(l.huge_content)
<class 'Proxy'>
>>> print(l.huge_content)
b'aaaa...aaaa'
```

We can even show you that we only read once from the stream:

```
>>> class PrintIO(io.BytesIO):
...     def read(self, size=-1):
...         print("Reading {} bytes from offset {}".format(size, self.tell()))
...         return super().read(size)
...
>>> l = LazyStructure.from_stream(PrintIO(b"a"*200))[0]
>>> print(l.huge_content)
Reading 200 bytes from offset 0
b'aaaa...aaaa'
>>> print(l.huge_content)
b'aaaa...aaaa'
```

Not all fields can be parsed lazily. For instance, a NULL-terminated *BytesField* must be parsed in its entirety before it knows its length. We need to know the field length if the field is followed by another field, so we must then still parse the field. In this case, the laziness of the field is ignored. To show this in action, see this example:

```
>>> class LazyLazyStructure(destructify.Structure):
...     field1 = destructify.BytesField(terminator=b'\0', lazy=True)
...     field2 = destructify.BytesField(terminator=b'\0', lazy=True)
...
>>> s = LazyLazyStructure.from_bytes(b"a\0b\0")
>>> type(s.field1), type(s.field2)
(<class 'bytes'>, <class 'Proxy'>)
```

Since the length of `field1` is required for parsing `field2`, we parse it regardless of the request to lazily parse it.

2.8 Combining offset with lazy

There is some important synergy between fields that have a offset set to an integer (i.e. do no depend on another field) and are lazy: this allows the field to be referenced during parsing, even if it is defined out-of-order:

```
>>> class SynergyStructure(destructify.Structure):
...     content = destructify.BytesField(length='length')
...     length = destructify.IntegerField(length=1, offset=-1, lazy=True)
...
>>> SynergyStructure.from_bytes(b"blahblah\x04")
<SynergyStructure: SynergyStructure(content=b'blah', length=4)>
```

This works because all lazy fields with lazy offsets are pre-populated in the parsing structure, making them being able to be referenced during parsing. In this example, the `length` field is referenced, therefore parsed and returned immediately and not through a Proxy object.

This is mostly to allow you to specify a structure that is more logical, though this structure would parse the same data:

```
class LessSynergyStructure(destructify.Structure):
    length = destructify.IntegerField(length=1, offset=-1)
    content = destructify.BytesField(length='length', offset=0)
```

CHAPTER 3

Custom fields

As part of the definition of a *Structure*, fields are used to interpret and write a small part of a binary structure. Each field is responsible for the following:

- Finding the start of the field relative to the previous field
- Consuming precisely enough bytes from a stream of bytes
- Converting these bytes to a Python representation
- Converting this back to a bytes representation
- Writing this back to a stream of bytes

3.1 Field idempotency

To ensure consistency across all fields, we have chosen to define two idempotency rules that holds for all built-in fields. Custom fields should attempt to adhere to these as well:

The idempotency of a field

When a value, that is written by a field, is read and written again by that same field, the byte representation must be the same.

When a value, that is read by a field, is written and read again by that same field, the Python representation must be the same.

What does it mean? In the most simple case, the byte and Python representation are linked to each other. This means, for instance, that writing `b'foo'` to a *BytesField*, will result in a `b'foo'` in the stream, and no other value has the same property.

In some cases, this does not hold. This is the case when different inputs converge to the same representation. For instance, considering a *VariableLengthIntegerField*, the byte representation of a value may be prepended with `0x80` bytes and they do not change the value of the field. So, when some other writer writes these pointless

bytes, Destructify has to ignore them. When writing a value, Destructify will then opt to write the least amount of bytes possible, meaning that the byte representation differs from the value that was read. However, Destructify can read this value again and it will be the same Python representation.

Similarly, a field may allow different types to be written to a stream. For instance, the *EnumField* allows you to write arbitrary values to *Field.to_stream*, but will always read them as `enum.Enum`, and also allows you to write this `enum.Enum` back to the stream.

All built-in fields will ensure that the two truths hold. If this is not possible, for instance due to alignment issues, an error will be raised. Some fields allow you to specify `strict=False`, which will disable these checks and may break idempotency.

3.2 Subclassing an existing field

If you only need to modify a field a little bit, you can probably come by with decoding/encoding-pairs (see *Decoding/encoding values*). Although these can be quite useful, they have one important limitation: you can't change the way the field reads and returns its value. Additionally, if you have to continuously write the same decoding/encoding-pair, this can become quite tiresome.

In the decoding/encoding example, we wrote a field that could be used to parse IPv4 addresses. Instead of repeating ourselves when we need to do this multiple times, we could also create an entirely new `IPAddressField`, setting the default for the *IntegerField.length* and changing the return value of the field:

```
import ipaddress

class IPAddressField(IntegerField):
    def __init__(self, *args, length=4, signed=False, **kwargs):
        super().__init__(*args, length=length, signed=signed, **kwargs)

    def from_stream(self, stream, context):
        value, length = super().from_stream(stream, context)
        return ipaddress.IPv4Address(value), length

    def to_stream(self, stream, value, context):
        return super().to_stream(stream, int(value), context)
```

Note how we have ordered the `super()` calls here: we want to read from the stream and then adjust the value, but we need to adjust the value before we are writing it to the stream.

Overriding *Field.from_stream()* and *Field.to_stream()* using Python inheritance is a common occurrence. Although the example above is very simple, you could adjust how the field works and acts entirely. For instance, the *BitField* is a subclass of `ByteField`, though it works on bits rather than bytes.

Note that there are many more functions you can override. The above example is a valid use-case, though overriding *Field.decode_value()* and *Field.encode_value()* might have been more appropriate. See *How a structure is read and written* for an overview of the methods where a value passes through to see where your use-case fits best. Also remember to read the documentation for *Field* to see what callbacks are used for what.

3.3 Writing your own field

The most complex method of changing how parsing works is by implementing your own field. You do this by inheriting from *Field* and implementing *Field.from_stream()* and *Field.to_stream()*. You then have full control over the stream cursor, how it reads values and how it returns those.

In this example, we'll be implementing variable-length quantities. Since this field has a variable-length (what's in a name) and parsing is entirely different from another field, we have to implement a new field.

Hint: A field implementing variable-length quantities is already in Destructify: *VariableLengthIntegerField*. You do not have to implement it yourself – this merely serves as an example.

The following code could be used to implement such a field:

```
class VariableLengthIntegerField(Field):
    def from_stream(self, stream, context):
        result = count = 0
        while True:
            count += 1
            c = stream.read(1)[0]  # TODO: verify that 1 byte is read
            result <<= 7
            result += c & 0x7f
            if not c & 0x80:
                break
        return result, count

    def to_stream(self, stream, value, context):  # TODO: check that value is positive
        result = [value & 0x7f]
        value >>= 7
        while value > 0:
            result.insert(0, value & 0x7f | 0x80)
            value >>= 7
        return stream.write(bytes(result))
```

Though actually parsing the field may seem like a complicated beast, the actual parsing is quite easy: you define how the field is read/written and you are done. When writing a field, you must always take care of the following:

- You must add in some checks to verify that everything is as you'd expect. In the above example, we have omitted these checks for brevity, but added a comment where you still need to add some checks, for instance, verify that we have not reached the end of the stream in *Field.from_stream()* and raise a `StreamExhaustedError`.
- You must ensure that the stream cursor is at the end of the field when you are done reading and writing. This is the place where the next field continues off. This is typically true, but if you need to look-ahead this may be an important gotcha.

There is more to implementing a field, as the next chapters will show you, though the basics will always remain the same. Read the full Python API for *Field* to see which callbacks are available.

3.4 Supporting length

You may have noticed that you can do `len(Structure)` on a structure and – if possible – get the byte length of the structure. This is actually implemented by calling `len(field)` on all fields in the structure. The default implementation of *Field* is to raise an `ImpossibleToCalculateLengthError`, so that when a field does not specify its length, the *Structure* that called will raise the same error.

Therefore, you are encouraged to add a `__len__` method to your fields when you can tell the length of a field beforehand (i.e. without a context):

```
class AlwaysFourBytesField(Field):
    def __len__(self):
        return 4
```

Note that you must return either a positive integer or raise an error. If your field depends on another field to determine its length, you should raise an error: you can only implement this field if you know its value regardless of the parsing state.

3.5 Supporting lazy read

The attribute *Field.lazy* controls how a field is read from the stream: if it is `True`, the field is not actually read during parsing, but only on its first access. This requires the field to know how much it needs to skip to find the start of the next field. This is implemented by *Field.seek_end()*, which is only called in the case that the start of the next field must be calculated (this is not the case e.g. if the next field has an absolute offset).

The default implementation is to check whether `len(field)` returns a usable result, and skips this amount of bytes. If the result is not usable, `None` is returned, and the field is read regardless of the *Field.lazy* setting.

However, there are cases where we can simply read a little bit of data to determine the length of the field, and then skip over the remainder of the field without parsing the entire field. This can be implemented by writing your own *Field.seek_end()*, which is more efficient than reading the entire field.

For instance, say that we have want to implement how UTF-8 encodes its length: if the first byte starts with `0b0`, it is a single byte-value, if the first byte starts with `0b110`, it is a two-byte value, `0b1110` a three-byte value and so forth. You could write a field like this:

```
class UTF8CharacterField(destructify.Field):
    def _get_length_from_first_byte(self, value):
        val = ord(value)
        for length, start_bits in enumerate(0b0, 0b110, 0b1110, 0b11110, 0b111110,␣
↪0b1111110):
            if val >> ((8 - start_bits.bit_length()) if start_bits else 7) == start_
↪bits:
                return length
        raise ParseError("Invalid start byte.")

    def seek_end(self, stream, context, offset):
        read = stream.read(1)
        if len(read) != 1:
            raise StreamExhaustedError()
        return stream.seek(self._get_length_from_first_byte(read) - 1, io.SEEK_CUR)

    def from_stream(self, stream, context):
        # left as an exercise to the reader

    def to_stream(self, stream, context):
        # left as an exercise to the reader
```

This still reads the first byte of the structure, but does not need to parse the entire structure.

3.6 Testing your field

Now, the only thing left is writing unittests for this. Since this field is mostly simple idempotent, we can use these simple tests to verify it all works according to plan, You may notice that the only simple idempotency exception is that

values may be repended with `80` bytes as that does not change its value:

```
class VariableLengthIntegerFieldTest(DestructifyTestCase):
    def test_basic(self):
        self.assertFieldStreamEqual(b'\x00', 0x00, VariableLengthIntegerField())
        self.assertFieldStreamEqual(b'\x7f', 0x7f, VariableLengthIntegerField())
        self.assertFieldStreamEqual(b'\x81\x00', 0x80, VariableLengthIntegerField())
        self.assertFieldFromStreamEqual(b'\x80\x80\x7f', 0x7f,␣
↪VariableLengthIntegerField())

    def test_negative_value(self):
        with self.assertRaises(OverflowError):
            self.call_field_to_stream(VariableLengthIntegerField(), -1)

    def test_stream_not_sufficient(self):
        with self.assertRaises(StreamExhaustedError):
            self.call_field_from_stream(VariableLengthIntegerField(), b'\x81\x80\x80')
```

CHAPTER 4

GUI & Hex Viewer

The Destructify GUI is a method to easily analyze raw binary data, and how it is handled by the structures you have defined.

Using the GUI is very easy:

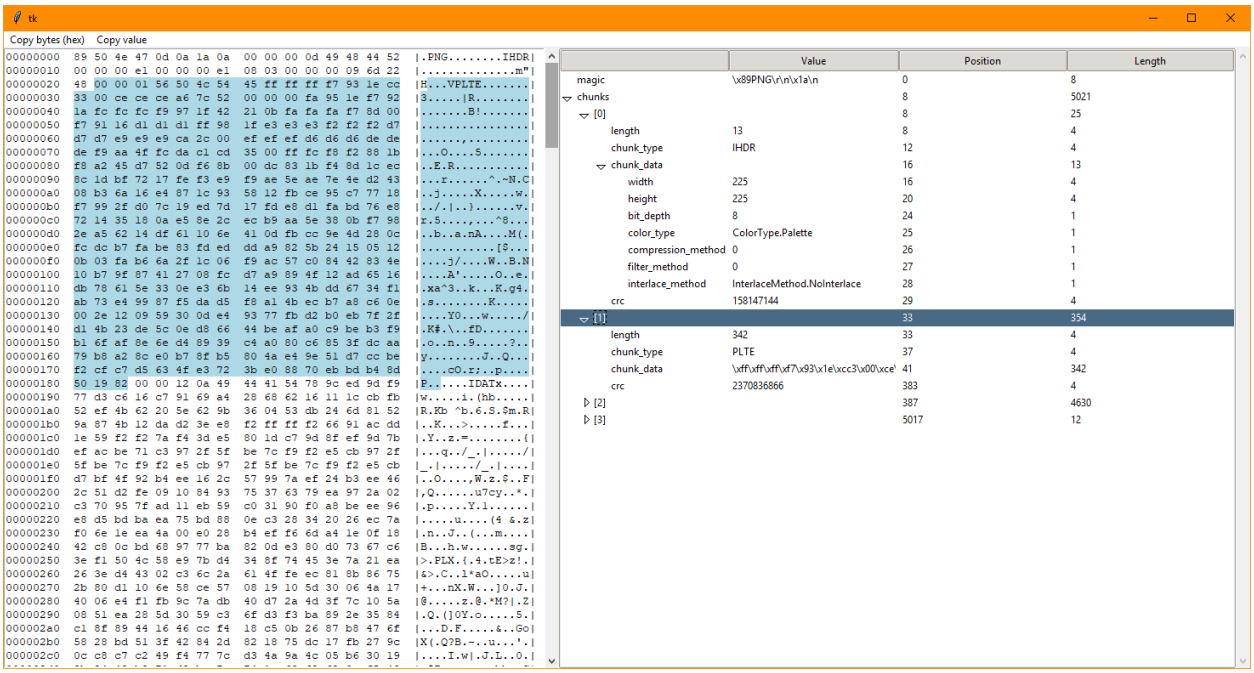
```
import destructify
from mylib import MyStructure

with open("mydata.bin", "rb") as f:
    destructify.gui.show(MyStructure, f)
```

You can also use the command-line launcher:

```
python -m destructify.gui mylib.MyStructure mydata.bin
```

Hint: It is best to provide a dotted path to the location where your structure resides. You can also use `-f` to provide a path to the source file containing the structure.

The following screenshot shows how this might look if you are parsing a PNG file:


tk
Copy bytes (hex)
Copy value
00000000 89 50 4e 47 0d 0a 1a 0a 00 00 00 0d 49 48 44 52 |.PNG........IHDR|
00000010 00 00 00 e1 00 00 00 e1 08 03 00 00 00 09 6d 22 |..............m"|
00000020 48 00 00 01 56 50 4c 54 45 ff ff ff f7 93 1e cc |H...VPLTE......|
00000030 33 00 ce ce ce a6 7c 52 00 00 00 fa 95 1e f7 92 |3.....|R.......|
00000040 1a fc fc fc f9 97 1f 42 21 0b fa fa fa f7 8d 00 |.......B!.......|
00000050 f7 91 16 d1 d1 d1 ff 98 1f e3 e3 e3 f2 f2 f2 d7 |................|
00000060 d7 d7 e9 e9 e9 ca 2c 00 ef ef ef d6 d6 d6 de de |......,.........|
00000070 de f9 aa 4f fc da c1 cd 35 00 ff fc f8 f2 88 1b |...O....5.......|
00000080 f8 a2 45 d7 52 0d f6 8b 00 dc 83 1b f4 8d 1c ec |..E.R...........|
00000090 8c 1d bf 72 17 fe f3 e9 f9 ae 5e ae 7e 4e d2 43 |...r......^.~N.C|
000000a0 08 b3 6a 16 e4 87 1c 93 58 12 fb ce 95 c7 77 18 |..j.....X.....w.|
000000b0 f7 99 2f d0 7c 19 ed 7d 17 fd e8 d1 fa bd 76 e8 |../.|..}......v.|
000000c0 72 14 35 18 0a e5 8e 2c ec b9 aa 5e 38 0b f7 98 |r.5....,...^8...|
000000d0 2e a5 62 14 df 61 10 6e 41 0d fb cc 9e 4d 28 0c |..b..a.nA...M(.|
000000e0 fc dc b7 fa be 83 fd ed dd a9 82 5b 24 15 05 12 |...........[$...|
000000f0 0b 03 fa b6 6a 2f 1c 06 f9 ac 57 c0 84 42 83 4e |....j/....W..B.N|
00000100 10 b7 9f 87 41 27 08 fc d7 a9 89 4f 12 ad 65 16 |....A'.....O..e.|
00000110 db 78 61 5e 33 0e e3 6b 14 ee 93 4b dd 67 34 f1 |.xa^3..k...K.g4.|
00000120 ab 73 e4 99 87 f5 da d5 f8 a1 4b ec b7 a8 c6 0e |.s........K.....|
00000130 00 2e 12 09 59 30 0d e4 93 77 fb d2 b0 eb 7f 2f |....Y0...w...../|
00000140 d1 4b 23 de 5c 0e d8 66 44 be af a0 c9 be b3 f9 |.K#.\..fD.......|
00000150 b1 6f af 8e 6e d4 89 39 c4 a0 80 c6 85 3f dc aa |.o..n..9.....?..|
00000160 79 b8 a2 8c e0 b7 8f b5 80 4a e4 9e 51 d7 cc be |y........J..Q...|
00000170 f2 cf c7 d5 63 4f e3 72 3b e0 88 70 eb bd b4 8d |....cO.r;..p....|
00000180 50 19 82 00 00 12 0a 49 44 41 54 78 9c ed 9d f9 |P.......IDATx....|
00000190 77 d3 c6 16 c7 91 69 a4 28 68 62 16 11 1c cb fb |w.....i.(hb.....|
000001a0 52 ef 4b 62 20 5e 62 9b 36 04 53 db 24 6d 81 52 |R.Kb ^b.6.S.$m.R|
000001b0 9a 87 4b 12 da d2 3e e8 f2 ff ff f2 66 91 ac dd |..K...>.....f...|
000001c0 1e 59 f2 f2 7a f4 3d e5 80 1d c7 9d 8f ef 9d 7b |.Y..z.=........{|
000001d0 ef ac be 71 c3 97 2f 5f be 7c f9 f2 e5 cb 97 2f |...q../_.|.....|
000001e0 5f be 7c f9 f2 e5 cb 97 2f 5f be 7c f9 f2 e5 cb |_.|...../_.|....|
000001f0 d7 bf 4f 92 b4 ee 16 2c 57 99 7a ef 24 b3 ee 46 |..O....,W.z.$..F|
00000200 2c 51 d2 fe 09 10 84 93 75 37 63 79 ea 97 2a 02 |,Q......u7cy..*.|
00000210 c3 70 95 7f ad 11 eb 59 c0 31 90 f0 a8 be ee 96 |.p.....Y.1......|
00000220 e8 d5 bd ba ea 75 bd 88 0e c3 28 34 20 26 ec 7a |.....u....(4 &.z|
00000230 f0 6e 1e ea 4a 00 e0 28 b4 ef f6 6d a4 1e 0f 18 |.n..J..(...m....|
00000240 42 c8 0c bd 68 97 77 ba 82 0d e3 80 d0 73 67 c6 |B...h.w......sg.|
00000250 3e f1 50 4c 58 e9 7b d4 34 8f 74 45 3e 7a 21 ea |>.PLX.{.4.tE>z!.|
00000260 26 3e d4 43 02 c3 6c 2a 61 4f fe ec 81 8b 86 75 |&>.C..l*aO.....u|
00000270 2b 80 d1 10 6e 58 ce 57 08 19 10 5d 30 06 4a 17 |+...nX.W...]0.J.|
00000280 40 06 e4 f1 fb 9c 7a db 40 d7 2a 4d 3f 7c 10 5a |@.....z.@.*M?|.Z|
00000290 08 51 ea 28 5d 30 59 c3 6f d3 f3 ba 89 2e 35 84 |.Q.(]0Y.o.....5.|
000002a0 c1 8f 89 44 16 46 cc f4 18 c5 0b 26 87 b8 47 6f |...D.F.....&..Go|
000002b0 58 28 bd 51 3f 42 84 2d 82 18 75 dc 17 fb 27 9c |X(.Q?B.-..u...'.|
000002c0 0c c8 c7 c2 49 f4 77 7c d3 4a 9a 4c 05 b6 30 19 |....I.w|.J.L..0.|
Value
Position
Length
magic
\x89PNG\r\n\x1a\n
0
8
chunks
8
5021
[0]
8
25
length
13
8
4
chunk_type
IHDR
12
4
chunk_data
16
13
width
225
16
4
height
225
20
4
bit_depth
8
24
1
color_type
ColorType.Palette
25
1
compression_method
0
26
1
filter_method
0
27
1
interlace_method
InterlaceMethod.NoInterlace
28
1
crc
158147144
29
4
[1]
33
354
length
342
33
4
chunk_type
PLTE
37
4
chunk_data
\xff\xff\xff\xf7\x93\x1e\xcc3\x00\xce'
41
342
crc
2370836866
383
4
[2]
387
4630
[3]
5017
12

CHAPTER 5

Python API

5.1 Structure

class destructify.**Structure**(*_context=None*, ***kwargs*)

You use *Structure* as the base class for the definition of your structures. It is a class with a metaclass of StructureBase that enables the fields to be parsed separately.

len(Structure)

This is a class method that allows you to retrieve the size of the structure, if possible.

classmethod from_stream(*stream*, *context=None*)

Reads a stream and converts it to a *Structure* instance. You can explicitly provide a *ParsingContext*, otherwise one will be created automatically.

This will seek over the stream if one of the alignment options is set, e.g. ParsingContext.alignment or *Field.offset*. The return value in this case is the difference between the start offset of the stream and the offset of the highest read byte. In most cases, this will simply equal the amount of bytes consumed from the stream.

Parameters

- **stream** – A buffered bytes stream.
- **context** (ParsingContext) – A context to use while parsing the stream.

Return type *Structure*, int

Returns A tuple of the constructed *Structure* and the amount of bytes read (defined as the last position of the read bytes).

classmethod from_bytes(*bytes*)

A short-hand method of calling *from_stream()*, using bytes rather than a stream, and returns the constructed *Structure* immediately.

classmethod initialize(*context*)

This classmethod allows you to modify the *ParsingContext*, just after all values were read from the

stream and *Field.get_initial_value()* was called, but before the *Structure* is created. This can be used to modify some values of the structure just before it is being created.

Parameters **context** (ParsingContext) – The context of the initializer

to_stream(*stream*, *context=None*)
Writes the current *Structure* to the provided stream. You can explicitly provide a *ParsingContext*, otherwise one will be created automatically.

This will seek over the stream if one of the alignment options is set, e.g. ParsingContext.alignment or *Field.offset*. The return value in this case is the difference between the start offset of the stream and the offset of the highest written byte. In most cases, this will simply equal the amount of bytes written to the stream.

Parameters

- **stream** – A buffered bytes stream.
- **context** (ParsingContext) – A context to use while writing the stream.

Return type int

Returns The number bytes written to the stream (defined as the maximum position of the bytes that were written)

to_bytes()
A short-hand method of calling *to_stream()*, writing to bytes rather than to a stream. It returns the constructed bytes immediately.

finalize(*context*)
Function that allows for modifying the *ParsingContext* just after filling the context with the values obtained by *Field.get_final_value()*, before it will be converted to binary data. This can be used to modify some values of the structure just before it is being written, e.g. for checksums.

Parameters **context** (ParsingContext) – The context of the finalizer

__bytes__()
Same as *to_bytes()*, allowing you to use bytes(structure)

classmethod as_cstruct()

_meta
This allows you to access the StructureOptions class of this *Structure*.

_context
If this *Structure* was created by *from_stream()*, this contains the *ParsingContext* that was used during the processing. Otherwise, this attribute is undefined.

5.2 Field

class destructify.**Field**(*, *name=None*, *default=NOT_PROVIDED*, *override=NOT_PROVIDED*, *decoder=None*, *encoder=None*, *offset=None*, *skip=None*, *lazy=False*)
A basic field is incapable of parsing or writing anything, as it is intended to be subclassed.

ctype
A friendly description of the field in the form of a C-style struct definition.

preparsable
Indicates whether this field is preparsable, i.e. the field is lazy and has an absolute offset set.

full_name
The full name of this *Field*.

field_context
The *FieldContext* that is used in the *ParsingContext* for this field. It returns a partially resolved function call with the current field already set.

Return type type

with_name(*name*)
Context manager that yields this *Field* with a different name. If *name* is None, this is ignored.

A *Field* also defines the following methods:

len(field)
You can call len on a field to retrieve its byte length. It can either return a value that makes sense, or it will raise an ImpossibleToCalculateLengthError when the length depends on something that is not known yet.

Some attributes may affect the length of the structure, while they do not affect the length of the field. This includes attributes such as *skip*. These are automatically added when the structure sums up all fields.

If you need to override how the structure sums the length of fields, you can override _length_sum. You must then manually also include those offsets. This is only used by *BitField*.

initialize()
Hook that is called after all fields on a structure are loaded, so some additional multi-field things can be arranged.

get_initial_value(*value*, *context*)
Returns the initial value given a context. This is used by *Structure.from_stream()* to retrieve the value that is read from the stream. It is called after all fields have been parsed, so inter-field dependencies can be resolved here.

The value may be a proxy object if *lazy* is set.

Parameters

- **value** – The value to retrieve the final value for.
- **context** (ParsingContext) – The context of this field.

get_final_value(*value*, *context*)
Returns the final value given a context. This is used by *Structure.to_stream()* to retrieve the value that is to be written to the stream. It is called before any fields have been processed, so inter-field dependencies can be resolved here.

Parameters

- **value** – The value to retrieve the final value for.
- **context** (ParsingContext) – The context of this field.

seek_start(*stream*, *context*, *offset*)
This is called before the field is parsed/written. It should expect the stream to be aligned to the ending of the previous field. It is intended to seek its starting position. This makes sense if the offset is set, for instance. In the case this stream is not tellable and no seek is performed, *offset* is returned unmodified.

Note that the *relative* offset is passed in, but the *absolute* offset is expected as a result.

Parameters

- **stream** (*io.BufferedIOBase*) – The IO stream to consume from.
- **context** (ParsingContext) – The context used for the parsing.

- **offset** (*int*) – The current relative offset in the stream

Returns The new absolute offset in the stream

seek_end(*stream*, *context*, *offset*)

This is called when the field is lazy and we need to find the end of the field. This is *not* called when the field is actually read, as *from_stream()* is expected to align to the end of the field.

This method should be as efficient as possible with retrieving the length. For instance, if it is possible to read a few bytes and then determine how long this field is, that is fine. If it is not possible without reading the entire field, this method should return `None`.

The default implementation is to call `len(self)` and use that if possible.

Parameters

- **stream** (*io.BufferedIOBase*) – The IO stream to consume from.
- **context** (`ParsingContext`) – The context used for the parsing.
- **offset** (*int*) – The current relative offset in the stream

Returns The new absolute offset in the stream, or None if this field can not be processed without parsing it entirely.

decode_value(*value*, *context*)

This value is called just after the value is retrieved from *from_stream()*. It should return an adjusted value that is the true representation of the value

Parameters

- **value** – The value to retrieve the decoded value for.
- **context** (`ParsingContext`) – The context of this field.

encode_value(*value*, *context*)

This value is called just before the value is passed to *to_stream()*. It should return an adjusted value that is accepted by *to_stream()*. This is typically used in conjunction with *encoder*.

Parameters

- **value** – The value to retrieve the encoded value for.
- **context** (`ParsingContext`) – The context of this field.

from_stream(*stream*, *context*)

Given a stream of bytes object, consumes a given bytes object to Python representation. The given stream is already at the start of the field. This method must ensure that the stream is after the end position of the field after reading. In other words, the following will typically hold true:

```
stream_at_start.tell() + result[1] == stream_at_end.tell()
```

The default implementation is to raise a `NotImplementedError` and subclasses must override this function.

Parameters

- **stream** (*io.BufferedIOBase*) – The IO stream to consume from. The current position is already set to the start position of the field.
- **context** (`ParsingContext`) – The context of this field.

Returns a tuple: the parsed value in its Python representation, and the amount of consumed bytes

to_stream(*stream*, *value*, *context*)
Writes a value to the stream, and returns the amount of bytes written. The given stream will already be at the start of the field, and this method must ensure that the stream cursor is after the end position of the field. In other words:

```
stream_at_start.tell() + result == stream_at_end.tell()
```

The default implementation is to raise a `NotImplementedError` and subclasses must override this function.

Parameters

- **stream** (*io.BufferedIOBase*) – The IO stream to write to.
- **value** – The value to write
- **context** (`ParsingContext`) – The context of this field.

Returns the amount of bytes written

decode_from_stream(*stream*, *context*)
Shortcut method to calling *from_stream()* and *decode_value()* in succession. Not intended to be overridden.

encode_to_stream(*stream*, *value*, *context*)
Shortcut method to calling *encode_value()* and *to_stream()* in succession. Not intended to be overridden.

5.3 ParsingContext

class destructify.**ParsingContext**(*structure=None*, ***, *parent=None*, *flat=False*, *stream=None*, *capture_raw=False*)
A context that is passed around to different methods during reading from and writing to a stream. It is used to contain context for the field that is being parsed.

While parsing, it is important to have some context; some fields depend on other fields during writing and during reading. The *ParsingContext* object is passed to several methods for this.

When using this module, you will get a *ParsingContext* when you define a property of a field that depends on another field. This is handled by storing all previously parsed fields in the context, or (if applicable) the *Structure* the field is part of. You can access this as follows:

```
context['field_name']
```

But, as a shorthand, you can also access it as an attribute of the *f* object:

```
context.f.field_name
```

context[key]
Returns the value of the specified *key*, either from the already parsed fields, or from the underlying structure, depending on the situation.

f
This object is typically used in `lambda` closures in *Field* declarations.

The *f* attribute allows you to access fields from this context, using attribute access. This is similar to using `context[key]`, but provides a little bit cleaner syntax. This object is separated from the scope of *ParsingContext* to avoid any name collisions with field names. (For instance, a field named `f` would be impossible to reach otherwise).

f.name
Access the current value of the named field in the *ParsingContext*, equivalent to `ParsingContext[name]`

f[name]
Alias for attribute access to allow accessing names that are dynamic or collide with the namespace (see below)

Two attributes are offered for parent and root access, and a third one to access the *ParsingContext*. These names still collide with field names you may want to specify, but the `f`-object is guaranteed to not add any additional name collisions in minor releases.

`f._`
Returns the *ParsingContext.f* attribute of the *ParsingContext.parent* object, so you can write `f.parent.parent.field`, which is equivalent to `context.parent.parent['field']`.

If you need to access a field named _, you must use `f['_']`

`f._root`
Returns the *ParsingContext.f* attribute of the *ParsingContext.root* object, so you can write `f.root.field`, which is equivalent to `context.root['field']`

If you need to access a field named `_root`, you must use `f['_root']`

`f._context`
Returns the actual *ParsingContext*. Used in cases where a `f`-object is only provided.

If you need to access a field named `_context`, you must use `f['_context']`

parent
Access to the parent context (useful when parsing a Structure inside a Structure). May be `None` if this is the uppermost context.

flat
Indicates that the parent context should be considered part of this context as well. This allows you to reference fields in both contexts transparently without the need of calling *parent*.

root
Retrieves the uppermost *ParsingContext* from this *ParsingContext*. May return itself.

fields
This is a dictionary of field names to *FieldContext*. You can use this to access information of how the fields were parsed. This is typically for debugging purposes, or displaying information about parsing structures.

done
Boolean indicating whether the parsing was done. If this is `True`, lazy fields can no longer become non-lazy.

field_values
Represents a immutable view on **all** field values from *fields*. This is highly inefficient if you only need to access a single value (use `context[key]`). The resulting dictionary is immutable.

This attribute is essentially only useful when constructing a new *Structure* where all field values are needed.

initialize_from_meta(*meta*, *structure=None*)
Adds fields to the context based on the provided StructureOptions. If *structure* is provided, the values in the structure are passed as values to the field contexts

When you are implementing a field yourself, you get a *ParsingContext* when reading from and writing to a stream.

5.4 FieldContext

class destructify.**FieldContext**(*field*, *context*, *value=NOT_PROVIDED*, ***, *field_name=None*, *parsed=False*, *offset=None*, *length=None*, *lazy=False*, *raw=None*)

This class contains information about the parsing state of the specified field.

field
The field this *FieldContext* applies to.

field_name
If set, this is the name of the field that is used in the context, regardless of what *field* has as *Field.name* set. If this is set, this is used with *Field.with_name()* when parsing lazily.

value
The current value of the field. This only makes sense when *has_value* is `True`. This can be a proxy object if *lazy* is true.

has_value
Indicates whether this field has a value. This is true only if the value is set or when *lazy* is true.

parsed
Indicates whether this field has been written to or read from the stream. This is also true when *lazy* is true.

resolved
Indicates whether this fields no longer requires stream access, i.e. it is parsed and *lazy* is false.

offset
Indicates the offset in the stream of this field. Is only set when *parsed* is true.

length
Indicates the length of this field. Is normally set when *parsed* is true, but may be not set when *lazy* is true and the length was not required to be calculated.

lazy
Indicates whether this field is lazily loaded. When a lazy field is resolved during parsing of the structure, i.e. while *ParsingContext.done* is false, resolving this field will affect *value*, *length* and set *lazy* to false. After *ParsingContext.done* has become true, these attributes will not be updated.

raw
If `ParsingContext.capture_raw` is true, this field will contain the raw bytes of the field.

subcontext
This may be set if the field created a subcontext to parse its inner field(s).

CHAPTER 6

Built-in fields specification

Destructify comes with a smorgasbord of built-in field types. This means that you can specify the most common structures right out of the box.

6.1 Common attributes

All fields are subclasses of *Field* and therefore come with some properties by default. These are the following and can be defined on every class:

`Field.`**`name`**
: The field name. This is set automatically by the *Structure*'s metaclass when it is initialized.

`Field.`**`default`**
: The field's default value. This is used when the *Structure* is initialized if it is provided. If it is not provided, the field determines its own default value.

  You can set it to one of the following:

  - A callable with zero arguments
  - A callable taking a *ParsingContext.f* object
  - A value

  All of the following are valid usages of the default attribute:

```
Field(default=None)
Field(default=3)
Field(default=lambda: datetime.datetime.now())
Field(default=lambda c: c.value)
```

  You can check whether a default is set using the `Field.has_default` attribute. The default given a context is obtained by calling `Field.get_default(context)`

`Field.`**`override`**
: Using *Field.override*, you can change the value of the field in a structure, just before it is being written to

a stream. This is useful if you, for instance, wish to override a field's value based on some other property in the structure. For instance, you can change a length field based on the actual length of a field.

You can set it to one of the following:

- A value
- A callable taking a *ParsingContext.f* object and the current value of the field

For instance:

```
Field(override=3)
Field(override=lambda c, v: c.value if v is None else v)
```

You can check whether an override is set using the `Field.has_override` attribute. The override given a context is obtained by calling `Field.get_overridden_value(value, context)`. Note, however, that you probably want to call *Field.get_final_value()* instead.

`Field.`**`decoder`**
`Field.`**`encoder`**

Sometimes, a field value can be different than the value in the binary structure. This can happen, for instance, if the value in the structure is off-by-one. Rather than overriding *Field.override* while writing, you can use *Field.encoder* and *Field.decoder* to change the way a value is written to and read from the stream, respectively.

You can set it to a callable taking the current value of the field:

```
Field(decoder=lambda v: v * 2, encoder=lambda v: v // 2)
```

The *Field.decoder* is used when reading from the stream. It is called from *Field.decode_value()*.

Field.encoder is used when writing to the stream. It is called from *Field.encode_value()*

`Field.`**`offset`**
`Field.`**`skip`**

The offset of the field absolutely in the stream (in the case of *offset*), or the offset of the field relative to the previous field (in the case of *skip*). *offset* can be a negative value to indicate an offset from the end of the stream.

You can't set both at the same time. You can set each to one of the following:

- A callable with zero arguments
- A callable taking a *ParsingContext.f* object
- A string that represents the field name that contains the value
- An integer

Fields are always processed in the order they are defined, so a field following a field that has one of these attributes set, will continue from the then-current position.

When you set *offset* or *skip*, *StructureOptions.alignment* is ignored for this field.

The value of *skip* is automatically accounted for when using `len(Structure)`. If *offset* is set, `len(Structure)` is not possible anymore.

`Field.`**`lazy`**

A lazy field is not parsed from the stream during the parsing of the bytes; its parsing is deferred until the value is evaluated. This is done by returning a Proxy object from the module lazy-object-proxy that references the offset of the field in the stream and the stream itself. The first time the Proxy object is evaluated, the stream is read and the data is parsed. This Proxy object can be used almost the same as an actual value.

This requires that the stream is not closed when not all lazy fields have been parsed. Additionally, the stream must be seekable to find the appropriate data.

Note that specifying *lazy* does not prohibit the parser to parse the field anyway, and return the actual value rather than a Proxy object. Some cases where this happens:

- The *lazy* attribute has no effect when a value can not be retrieved lazily, i.e. *Field.seek_end()* returns `None`, and the next field defines no absolute *offset*. In this case, the field must still be parsed to retrieve its full length, and is therefore parsed immediately.
- When *lazy* fields are referenced and subsequently parsed during parsing, the *Structure* will be built with the actual value rather than the Proxy object.

Additionally, *lazy* fields that have an absolute *offset* set (to an integer value), can be referenced during parsing, even if they are defined later.

This attribute has no effect when writing to a stream; a lazy value will be resolved by *Structure.to_stream()*.

6.2 BytesField

class destructify.**BytesField**(**args*, *length=None*, *terminator=None*, *step=1*, *terminator_handler='consume'*, *strict=True*, *padding=None*, ***kwargs*)

A *BytesField* can be used to read bytes from a stream. This is most commonly used as a base class for other methods, as it can be used for the most common use cases.

There are three typical ways to use this field:

- Setting a *BytesField.length* to read a specified amount of bytes from a stream.
- Setting a *BytesField.terminator* to read until the specified byte from a stream.
- Setting both *BytesField.length* and *BytesField.terminator* to first read the specified amount of bytes from a stream and then find the terminator in this amount of bytes.

length

This specifies the length of the field. This is the amount of data that is read from the stream and written to the stream. The length may also be negative to indicate an unbounded read, i.e. until the end of stream.

You can set this attribute to one of the following:

- A callable with zero arguments
- A callable taking a *ParsingContext.f* object
- A string that represents the field name that contains the length
- An integer

For instance:

```
class StructureWithLength(Structure):
    length = UnsignedByteField()
    value = BytesField(length='length')
```

The length given a context is obtained by calling `FixedLengthField.get_length(value, context)`.

When the class is initialized on a *Structure*, and the length property is specified using a string, the default implementation of the *Field.override* on the named attribute of the *Structure* is changed to match the length of the value in this *Field*.

Continuing the above example, the following works automatically:

```
>>> bytes(StructureWithLength(value=b"123456"))
b'\x06123456'
```

However, explicitly specifying the length would override this:

```
>>> bytes(StructureWithLength(length=1, value=b"123456"))
b'\x01123456'
```

This behaviour can be changed by manually specifying a different *Field.override* on `length`.

strict
: This boolean (defaults to `True`) enables raising errors in the following cases:

  - A `StreamExhaustedError` when there are not sufficient bytes to completely fill the field while reading.
  - A `StreamExhaustedError` when the terminator is not found while reading.
  - A `WriteError` when there are not sufficient bytes to fill the field while writing and *padding* is not set.
  - A `WriteError` when the field must be padded, but the bytes that are to be written are not a multiple of the size of *padding*.
  - A `WriteError` when there are too many bytes to fit in the field while writing.
  - A `WriteError` when the terminator is missing from the value, when using the *terminator_handler* `include`

  Disabling *BytesField.strict* is not recommended, as this may cause inadvertent errors.

padding
: When set, this value is used to pad the bytes to fill the entire field while writing, and chop this off the value while reading. Padding is removed right to left and must be aligned to the end of the value (which matters for multibyte paddings).

  While writing in *strict* mode, and the remaining bytes are not a multiple of the length of this value, a `WriteError` is raised. If *strict* mode is not enabled, the padding will simply be appended to the value and chopped of whenever required. However, this can't be parsed back by Destructify (as the padding is not aligned to the end of the structure).

  This can only be set when *length* is used.

terminator
: The terminator to read until. It can be multiple bytes.

  When this is set, *padding* is ignored while reading from a stream, but may be used to pad bytes that are written.

step
: The size of the steps for finding the terminator. This is useful if you have a multi-byte terminator that is aligned. For instance, when reading NULL-terminated UTF-16 strings, you'd expect two NULL bytes aligned to two bytes (from the start). Defaults to 1.

  Example usage:

```
>>> class TerminatedStructure(Structure):
...     foo = BytesField(terminator=b'\0')
...     bar = BytesField(terminator=b'\r\n')
...
```

(continues on next page)

(continued from previous page)

```
>>> TerminatedStructure.from_bytes(b"hello\0world\r\n")
<TerminatedStructure: TerminatedStructure(foo=b'hello', bar=b'world')>
```

terminator_handler
A string defining what to do with the terminator as soon as it is encountered. You have three options:

consume This is the default handler, and consumes the terminator, leaving it off the resulting value.

include This handler will include the entire terminator into the resulting value. You must also write it back yourself.

until This handler is only available when you are not using *length*, allowing you to consume up until, but not including the terminator. This means that the next field will include the terminator.

This class can be used trivially to extend functionality. For instance, *StringField* is a subclass of this field.

6.2.1 FixedLengthField

class destructify.**FixedLengthField**(*length*, **args*, ***kwargs*)
This class is identical to *BytesField*, but specifies the length as a required first argument. It is intended to read a fixed amount of *BytesField.length* bytes.

6.2.2 TerminatedField

class destructify.**TerminatedField**(*terminator=b'x00'*, **args*, ***kwargs*)
This class is identical to *BytesField*, but specifies the terminator as its first argument, defaulting to a single NULL-byte. It is intended to continue reading until *BytesField.terminator* is hit.

6.3 StringField

class destructify.**StringField**(**args*, *encoding=None*, *errors='strict'*, ***kwargs*)
The *StringField* is a subclass of *BytesField* that converts the resulting `bytes` object to a `str` object, given the *encoding* and *errors* attributes.

See *BytesField* for all available attributes.

encoding
The encoding of the string. This defaults to the value set on the `StructureOptions`, which defaults to `utf-8`, but can be any encoding supported by Python.

errors
The error handler for encoding/decoding failures. Defaults to Python's default of `strict`.

6.4 IntegerField

class destructify.**IntegerField**(*length*, *byte_order=None*, **args*, *signed=False*, ***kwargs*)
The *IntegerField* is used for fixed-length representations of integers.

Note: The *IntegerField* is not to be confused with the `IntField`, which is based on *StructField*.

---

length
The length (in bytes) of the field. When writing a number that is too large to be held in this field, you will get an `OverflowError`.

byte_order
The byte order (i.e. endianness) of the bytes in this field. If you do not specify this, you must specify a `byte_order` on the structure.

signed
Boolean indicating whether the integer is to be interpreted as a signed or unsigned integer.

6.5 VariableLengthIntegerField

class destructify.**VariableLengthIntegerField**(*, *name=None*, *default=NOT_PROVIDED*, *override=NOT_PROVIDED*, *decoder=None*, *encoder=None*, *offset=None*, *skip=None*, *lazy=False*)
Implementation of a variable-length quantity structure.

6.6 BitField

class destructify.**BitField**(*length*, **args*, *realign=False*, ***kwargs*)
A subclass of *FixedLengthField*, reading bits rather than bytes. The field writes and reads integers.

When using the *BitField*, you must be careful to align the field to whole bytes. You can use multiple *BitField* s consecutively without any problem, but the following would raise errors:

```
class MultipleBitFields(Structure):
    bit0 = BitField(length=1)
    bit1 = BitField(length=1)
    byte = FixedLengthField(length=1)
```

You can fix this by ensuring all consecutive bit fields align to a byte in total, or, alternatively, you can specify *realign* on the last *BitField* to realign to the next byte.

length
The amount of bits to read.

realign
This specifies whether the stream must be realigned to entire bytes after this field. If set, after bits have been read, bits are skipped until the next whole byte. This means that the intermediate bits are ignored. When writing and this boolean is set, it is padded with zero-bits until the next byte boundary.

Note that this means that the following:

```
class BitStructure(Structure):
    foo = BitField(length=5, realign=True)
    bar = FixedLengthField(length=1)
```

Results in this parsing structure:

```
76543210  76543210
fffff     bbbbbbbb
```

Thus, ignoring bits 2-0 from the first byte.

A *BitField* has some important gotchas and exceptions to normal fields:

- *StructureOptions.alignment* is *ignored* when two *BitField* follow each other, and the previous field does not specify *realign*.
- *Field.skip* and *Field.offset* must be specified in entire bytes, and require the field to be aligned.
- *Field.lazy* does not work, due to complexities with parsing partial bytes.
- `len(BitField)` returns the value in *bits* rather than in *bytes*. `len(Structure)` works properly, but requires that all fields are aligned, including the last field.

6.7 ConstantField

class destructify.**ConstantField**(*value*, *base_field=None*, **args*, ***kwargs*)

The *ConstantField* is intended to read/write a specific magic string from and to a stream. If anything else is read or written, an exception is raised. Note that the *Field.default* is also set to the magic.

value

The magic bytes that must be checked against.

base_field

The field to read the *value* from. If this is not set, and *value* is a bytes object, a *FixedLengthField* as its default. If the value is of any other object, you must specify this yourself.

6.8 StructField

class destructify.**StructField**(*format=None*, *byte_order=None*, **args*, *multibyte=True*, ***kwargs*)

The *StructField* enables you to use Python `struct` constructs if you wish to. Note that using complex formats in this field kind-of defeats the purpose of this module.

format

The format to be passed to the `struct` module. See Struct Format Strings in the manual of Python for information on how to construct these.

You do not need to include the byte order in this attribute. If you do, it acts as a default for the *byte_order* attribute if you do not specify one.

byte_order

The byte order to use for the struct. If this is not specified, and none is provided in the *format* field, it defaults to the `byte_order` specified in the meta of the `destructify.structures.Structure`.

multibyte

When set to `False`, the Python representation of this field is the first result of the tuple as returned by the `struct` module. Otherwise, the tuple is the result.

6.8.1 Subclasses of StructField

This project also provides several default implementations for the different types of structs. For each of the formats described in Struct Format Strings, there is a single-byte class. Note that you must specify your own

Each of the classes is listed in the table below.

Hint: Use a *IntegerField* when you know the amount of bytes you need to parse. Classes below are typically used for system structures and the *IntegerField* is typically used for network structures.

Base class	Format
CharField	c
ByteField	b
UnsignedByteField	B
BoolField	?
ShortField	h
UnsignedShortField	H
IntField	i
UnsignedIntField	I
LongField	l
UnsignedLongField	L
LongLongField	q
UnsignedLongLongField	Q
SizeField	n
UnsignedSizeField	N
HalfPrecisionFloatField	e
FloatField	f
DoubleField	d

6.9 StructureField

class destructify.**StructureField**(*structure*, **args*, *length=None*, ***kwargs*)

The *StructureField* is intended to create a structure that nests other structures. You can use this for complex structures, or when combined with for instance an *ArrayField* to create arrays of structures, and when combined with *SwitchField* to create type-based structures.

structure

The *Structure* class that is initialized for the sub-structure.

length

The length of this structure. This allows you to limit the structure's length. This is particularly useful when you have a *Structure* that contains an unbounded read, but the encapsulating structure limits this.

- A callable with zero arguments
- A callable taking a *ParsingContext.f* object
- A string that represents the field name that contains the size
- An integer

When specified using a string, this field does *not* override the value of the referenced field due to complications in calculating the length.

During reading and writing, if the specified length is larger than the structure, the remaining bytes are skipped. If it is shorter, the structure parsing will break.

Example usage:

```
>>> class Sub(Structure):
...     foo = FixedLengthField(length=11)
...
>>> class Encapsulating(Structure):
...     bar = StructureField(Sub)
...
>>> s = Encapsulating.from_bytes(b"hello world")
>>> s
<Encapsulating: Encapsulating(bar=<Sub: Sub(foo=b'hello world')>)>
>>> s.bar
<Sub: Sub(foo=b'hello world')>
>>> s.bar.foo
b'hello world'
```

This field providesthe *ParsingContext* of the substructure in *FieldContext.subcontext*.

6.10 ArrayField

class destructify.**ArrayField**(*base_field*, *count=None*, *length=None*, *until=None*, **args*, ***kwargs*)

A field that repeats the provided base field multiple times. The implementation will build a structure-like parsing context with field names that are the element indexes.

base_field
The field that is to be repeated.

count
This specifies the amount of repetitions of the base field.

You can set it to one of the following:

- A callable with zero arguments
- A callable taking a *ParsingContext.f* object
- A string that represents the field name that contains the size
- An integer

The count given a context is obtained by calling `ArrayField.get_count(value, context)`.

When this attribute is set using a string, and the referenced field does not have an override set, the override of this field will be set to take the length of the value of this field.

When writing, the count must exactly match the amount of items in the provided iterable.

Example usage:

```
>>> class ArrayStructure(Structure):
...     count = UnsignedByteField()
...     foo = ArrayField(TerminatedField(terminator=b'\0'), count='count')
...
>>> s = ArrayStructure.from_bytes(b"\x02hello\0world\0")
>>> s.foo
[b'hello', b'world']
```

length
This specifies the size of the field, if you do not know the count of the fields, but do know the size.

You can set it to one of the following:

- A callable with zero arguments
- A callable taking a *ParsingContext.f* object
- A string that represents the field name that contains the size
- An integer

The length given a context is obtained by calling `ArrayField.get_length(value, context)`.

You can specify a negative length if you want to read until the stream ends. Note that this is currently implemented by swallowing a `StreamExhaustedError` from the base field.

When specified using a string, this field does *not* override the value of the referenced field due to complications in calculating the length.

When writing using a positive length, the written amount of bytes must be exactly the specified length.

until
This is a function taking a context and the value of the most-recent parsed element. If this function returns true, the parsing stops.

This function is ignored during writing.

6.11 ConditionalField

class destructify.**ConditionalField**(*base_field*, *condition*, **args*, *fallback=None*, ***kwargs*)
A field that may or may not be present. When the *condition* evaluates to true, the *base_field* field is parsed, otherwise the field is `None`.

base_field
The field that is conditionally present.

condition
This specifies the condition on whether the field is present.

You can set it to one of the following:

- A callable with zero arguments
- A callable taking a *ParsingContext.f* object
- A string that represents the field name that evaluates to true or false. Note that `b'\0'` evaluates to true.
- A value that is to be evaluated

The condition given a context is obtained by calling `ConditionalField.get_condition(value, context)`.

fallback
The value that is used in the structure when loading from the stream and no value was present in the stream. Defaults to `None`, but could be any value.

6.12 SwitchField

class destructify.**SwitchField**(*cases*, *switch*, **args*, *other=None*, ***kwargs*)
The *SwitchField* can be used to represent various types depending on some other value. You set the different cases using a dictionary of value-to-field-types in the *cases* attribute. The *switch* value defines the case that is applied. If none is found, an error is raised, unless *other* is set.

cases
A dictionary of all cases mapping to a specific *Field*.

switch
This specifies the switch, i.e. the key for *cases*.

You can set it to one of the following:

- A callable with zero arguments
- A callable taking a *ParsingContext.f* object
- A string that represents the field name that evaluates to the value of the condition
- A value that is to be evaluated

other
The 'default' case that is used when the *switch* is not part of the *cases*. If not specified, and an unknown value is encountered, an exception is raised.

Hint: A confusion is easily made by setting *Field.default* instead of *other*, though their purposes are entirely different.

Example:

```
class ConditionalStructure(Structure):
    type = EnumField(IntegerField(1), enum=Types)
    perms = SwitchField(cases={
        Types.FIRST: StructureField(Structure1),
        Types.SECOND: StructureField(Structure2),
    }, other=StructureField(Structure0), switch='type')
```

6.13 EnumField

class destructify.**EnumField**(*base_field*, *enum*, **args*, ***kwargs*)
A field that takes the value as evaluated by the *base_field* and parses it as the provided *enum*.

While writing, the value can be of a enum member of specified *enum*, a string referencing an enum member, or the value that is to be written. Note that providing a string that is not a valid enum member, will be passed to the field directly.

During parsing, a value must be a valid enum member, or the enum must properly handle the case of missing members.

base_field
The field that returns the value that is provided to the `enum.Enum`

enum
The `enum.Enum` class.

You can also use an *EnumField* to handle flags:

```
>>> class Permissions(enum.IntFlag):
...     R = 4
...     W = 2
...     X = 1
...
```

(continues on next page)

(continued from previous page)

```
>>> class EnumStructure(Structure):
...     perms = EnumField(UnsignedByteField(), enum=Permissions)
...
>>> EnumStructure.from_bytes(b"\x05")
<EnumStructure: EnumStructure(perms=<Permissions.R|X: 5>)>
```

CHAPTER 7

Version history

7.1 Releases

7.1.1 v0.2.0 (2019-03-23)

This release adds more field types and further improves on existing code. It also extends the documentation significantly.

- Added Destructify GUI, contributed by mvdnes.
- Added *StructureOptions.encoding*
- Added *StructureOptions.alignment*, *Field.offset* and *Field.skip*, implemented by *Field.seek_start*
- Added *Field.lazy*
- Added *Field.decoder*, *Field.encoder* and *Structure.initialize()*
- Added *BytesField.terminator_handler*
- Added *ConditionalField.fallback*
- Added *ArrayField.until*
- New field *BytesField*, merging the features of *FixedLengthField* and *TerminatedField*. These fields will remain as subclasses.
- New field: *ConstantField*
- New field: *SwitchField*
- New field: *VariableLengthIntegerField*
- Merged `FixedLengthStringField` and `TerminatedStringField` into *StringField*
- Removed hook functions `Field.from_bytes()` and `Field.to_bytes()`
- Removed all byte-order specific subclasses from *StructField*.

- Add option to *ParsingContext* to capture the raw bytes, available in *ParsingContext.fields*
- Add *ParsingContext.fields* for information about the parsing structure.
- Added *ParsingContext.f* for raw attribute access; this is now passed to lambdas.
- Added *this* for quick construction of lambdas
- `Substream` is now a wrapper instead of a full-fetched BufferedReader
- Numerous bugfixes for consistent building of fields.

7.1.2 v0.1.0 (2019-02-17)

This release features several new field types, and bugfixes from the previous release. Also some backwards-incompatible changes were made.

- Added *StructureOptions.byte_order*
- Added *Structure.as_cstruct()*
- Added `Structure.__len__()`
- Added `Structure.full_name()`
- *FieldContext* is now *ParsingContext*
- New field: *ConditionalField*
- New field: *EnumField*
- New field: *BitField*
- New field: *IntegerField*, renamed struct-based field to `IntField`
- New field: `FixedLengthStringField`
- New field: `TerminatedStringField`
- Support strict, negative lengths and padding in `structify.fields.FixedLengthField`
- Support length in `structify.fields.ArrayField`, renamed `ArrayField.size` to *ArrayField.count*
- Support step `structify.fields.TerminatedField`
- Fixed `structify.fields.StructureField` to use `structify.Substream`
- Fixed double-closing a `structify.Substream`

7.1.3 v0.0.1 (2018-04-07)

Initial release.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index

Symbols

A

B

C

D

E

F

G