Construct Documentation

Release 2.9

Arkadiusz Bulski

May 18, 2018

Contents

CHAPTER 1

User Guide

1.1 Introduction

Construct is a powerful **declarative** and **symmetrical** parser and builder for binary data.

Instead of writing *imperative code* to parse a piece of data, you declaratively define a *data structure* that describes your data. As this data structure is not code, you can use it in one direction to *parse* data into Pythonic objects, and in the other direction, to *build* objects into binary data.

The library provides both simple, atomic constructs (such as integers of various sizes), as well as composite ones which allow you form hierarchical and sequential structures of increasing complexity. Construct features **bit and byte granularity**, easy debugging and testing, an **easy-to-extend subclass system**, and lots of primitive constructs to make your work easier:

- Fields: raw bytes or numerical types
- Structs and Sequences: combine simpler constructs into more complex ones
- Bitwise: splitting bytes into bit-grained fields
- Adapters: change how data is represented
- Arrays/Ranges: duplicate constructs
- Meta-constructs: use the context (history) to compute the size of data
- If/Switch: branch the computational path based on the context
- On-demand (lazy) parsing: read and parse only the fields what you require
- Pointers: jump from here to there in the data stream
- Tunneling: prefix data with a byte count or compress it

1.1.1 Example

A `Struct` is a collection of ordered, named fields:

```
>>> format = Struct(
...     "signature" / Const(b"BMP"),
...     "width" / Int8ub,
...     "height" / Int8ub,
...     "pixels" / Array(this.width * this.height, Byte),
... )
>>> format.build(dict(width=3,height=2,pixels=[7,8,9,11,12,13]))
b'BMP\x03\x02\x07\x08\t\x0b\x0c\r'
>>> format.parse(b'BMP\x03\x02\x07\x08\t\x0b\x0c\r')
Container(signature=b'BMP')(width=3)(height=2)(pixels=[7, 8, 9, 11, 12, 13])
```

A `Sequence` is a collection of ordered fields, and differs from `Array` and `GreedyRange` in that those two are homogenous:

```
>>> format = Sequence(PascalString(Byte, "utf8"), GreedyRange(Byte))
>>> format.build([u"lalaland", [255,1,2]])
b'\nlalaland\xff\x01\x02'
>>> format.parse(b"\x004361789432197")
['', [52, 51, 54, 49, 55, 56, 57, 52, 51, 50, 49, 57, 55]]
```

Construct has been used to parse:

- Networking formats like Ethernet, IP, ICMP, IGMP, TCP, UDP, DNS, DHCP
- Binary file formats like Bitmaps, PNG, GIF, EMF, WMF
- Executable binaries formats like ELF32, PE32
- Filesystem layouts like Ext2, Fat16, MBR

See more examples in current gallery and in deprecated gallery.

1.1.2 Development and support

Please use github issues to ask general questions, make feature requests (and vote for them), report issues and bugs, and to submit PRs. Feel free to request any changes that would support your project. There is also a gitter chat but using Issues is highly recommended.

Main documentation is at readthedocs, which is substantial. Source is at github. Releases are available at pypi.

1.1.3 Requirements

Construct should run on CPython 2.7 3.3 3.4 3.5 3.6 3.7 and PyPy 2.7 3.5 implementations. Recommended is CPython 3.6 and PyPy (any version) because they support ordered keyword arguments, and also PyPy achieves much better performance. Therefore PyPy would be most recommended.

Following modules are needed only if you want to use certain features:

- Enum34 is optional if you want Enum EnumFlags to take labels from IntEnum IntFlag.
- Numpy is optional, if you want to serialize arrays using Numpy protocol. Otherwise arrays can still be serialized using PrefixedArray.
- Arrow is optional, if you want to use Timestamp class.
- Different Python versions support different compression modules (like gzip lzma), if you want to use Compressed class.
- Ruamel.yaml is optional, if you want to use KaitaiStruct (KSY) exporter.

1.1.4 Installing

The library is downloadable and installable from Pypi. Just use standard command-line. There are no hard dependencies, but if you would like to install all supported (not required) modules listed above, you can use the 2nd command-line form.

- pip install construct
- pip install construct[extras]

1.2 Transition to 2.8

1.2.1 Overall

All fields and complex constructs are now nameless, you need to use / operator to name them. Look at Struct Sequence Range for how to use + >> [] operators to construct larger instances.

Integers and floats

{U,S}{L,B,N}Int{8,16,24,32,64} was made Int{8,16,24,32,64}{u,s}{l,b,n}

Byte, Short, Int, Long were made aliases to Int{8,16,32,64}ub

{B,L,N}Float{32,64} was made Float{32,64}{b,l,n}

Single, Double were made aliases to Float{32,64}b

VarInt was added

Bit, Nibble, Octet remain

All above were made singletons

Fields

Field was made Bytes (operates on b-strings)

BytesInteger was added (operates on integers)

BitField was made BitsInteger (operates on integers)

GreedyBytes was added

Flag was made a singleton

Enum takes the *default* keyword argument (no underscores)

Enum was fixed, the context value is a string label (not integer).

FlagsEnum remains

Strings

String remains

PascalString argument *length_field=UBInt8* was made *lengthfield* and explicit

CString dropped *char_field*

GreedyString dropped *char_field*

All above use optional *encoding* argument or use global encoding (see `setglobalstringencoding()`)

Structures and Sequences

Struct uses syntax like `Struct("num"/Int32ub, "text"/CString())` and `"num"/Int32ub + "text"/CString()`

Sequence uses syntax like `Byte >> Int16ul` and `Sequence(Byte, Int16ul)`

On Python 3.6 you can also use syntax like `Struct(num=Int32ub, text=CString())` and `Sequence(num=Int32ub, text=CString())`

Ranges and Arrays

Array uses syntax like `Byte[10]` and `Array(10, Byte)`

Range uses syntax like `Byte[5:]` and `Byte[:5]` and `Range(min=5, max=2**64, Byte)`

GreedyRange uses syntax like `Byte[:]` and `GreedyRange(Byte)`

PrefixedArray takes explicit *lengthfield* before subcon

OpenRange and GreedyRange were dropped

OptionalGreedyRange was renamed to GreedyRange

RepeatUntil takes 3-argument (last element, list, context) lambda

Lazy collections

LazyStruct LazyRange LazySequence were added

OnDemand returns a parameterless lambda that returns the parsed object

OnDemandPointer was dropped

LazyBound remains

Padding and Alignment

Aligned takes explicit *modulus* before the subcon

Padded was added, also takes explicit *modulus* before the subcon

Padding remains

Padding and Padded dropped *strict* parameter

Optional

If dropped *elsevalue* and always returns None

IfThenElse parameters renamed to *thensubcon* and *elsesubcon*

Switch remains

Optional remains

Union takes explicit *parsefrom* so parsing seeks stream by selected subcon size, or does not seek by default

Select remains

Miscellaneous and others

Value was made Computed

Embed was made Embedded

Alias was removed

Magic was made Const

Const has reordered parameters, like `Const(b"\\x00")` and `Const(0, Int8ub)`.

Pass remains

Terminator was renamed Terminated

OneOf and NoneOf remain

Filter added

LengthValueAdapter was made Prefixed, and gained *includelength* option

Hex added

HexDumpAdapter was made HexDump

HexDump builds from hexdumped data, not from raw bytes

SlicingAdapter and IndexingAdapter were made Slicing and Indexing

ExprAdapter ExprSymmetricAdapter ExprValidator were added or remain

SeqOfOne was replaced by FocusedSeq

Numpy added

NamedTuple added

Check added

Error added

Default added

Rebuild added

StopIf added

Stream manipulation

Bitwise was reimplemented using Restreamed

Bytewise was added

Restreamed and Rebuffered were redesigned

Anchor was made Tell and a singleton

Seek was added

Pointer remains, size cannot be computed

Peek dropped *perform_build* parameter, never builds

Tunneling

RawCopy was added, returns both parsed object and raw bytes consumed

Prefixed was added, allows to put greedy fields inside structs and sequences

ByteSwapped and BitsSwapped were added

Checksum was added

Compressed was added

Exceptions

FocusedError OverwriteError were removed

FieldError was replaced with StreamError (raised when stream returns less than requested amount) and FormatFieldError (raised by FormatField class, for example if building Float from non-float value and struct.pack complaining).

1.3 Transition to 2.9

Warning: Construct is undergoing heavy changes at the moment, expect unstable API until further notice.

1.3.1 Overall

Compilation feature for faster performance! Read this tutorial chapter, particularly its restrictions section.

Docstrings of all classes were overhauled. Check the Core API pages.

General classes

All constructs: *parse build sizeof* methods take context entries ONLY as keyword parameters **contextkw (see tutorial page)

All constructs: *parse_file* and *build_file* methods were added (see tutorial page)

All constructs: operator * can be used for docstrings and parsed hooks (see tutorial page)

All constructs: added *compile* and *benchmark* methods (see tutorial page)

All constructs: added *parsed* hook/callback (see tutorial page)

Compiled added (used internally)

String* require explicit encodings, all of them support UTF16 UTF32 encodings, but PaddedString CString dropped some parameters and support only encodings explicitly listed in *possiblestringencodings* (see tutorial page)

PaddedString CString classes reimplemented using NullTerminated NullStripped

String* build empty strings into empty bytes (despite for example UTF16 encoding empty string into 2 bytes marker)

String class renamed to PaddedString

Enum FlagsEnum can merge labels from IntEnum IntFlag, from enum34 module (see tutorial page)

Enum FlagsEnum dropped *default* parameter but returns integer if no mapping found (see tutorial page)

Enum FlagsEnum can build from integers and labels, and expose labels as attributes, as bitwisable strings (see tutorial page)

FlagsEnum had parsing semantics fixed (affecting multi-bit flags)

Mapping replaced SymmetricMapping, and dropped *default* parameter (see API page)

Struct Sequence FocusedSeq Union LazyStruct have new embedding semantics (see tutorial page)

Struct Sequence FocusedSeq Union LazyStruct are exposing subcons, as attributes and in _subcons context entry (see tutorial page)

Struct Sequence FocusedSeq Union LazyStruct are exposing _ _params _root _parsing _building _sizing _subcons _io _index entries in the context (see tutorial page)

EmbeddedBitStruct removed, instead use BitStruct with Bytewise-wrapped fields (see tutorial page)

Array reimplemented without Range, does not use stream.tell()

Range removed, GreedyRange does not support [:] syntax

Array GreedyRange RepeatUntil added *discard* parameter (see tutorial page)

Const has reordered parameters, *value* before *subcon* (see API page)

Index added, in Miscellaneous (see tutorial page)

Pickled added, in Miscellaneous (see tutorial page)

Timestamp added, in Miscellaneous (see tutorial page)

Hex HexDump reimplemented, return bytes and not hexlified strings (see tutorial page)

Select dropped *includename* parameter (see API page)

If IfThenElse parameter *predicate* renamed to *condfunc*, and cannot be embedded (see API page)

Switch updated, *default* parameter is *Pass* instead of *NoDefault*, dropped *includekey* parameter, and cannot be embedded (see API page)

EmbeddedSwitch added, in Conditional (see tutorial page)

StopIf raises *StopFieldError* instead of *StopIteration* (see API page)

Pointer changed size to 0, can be parsed lazily, can also select a stream from context entry

PrefixedArray parameter *lengthfield* renamed to *countfield* (see API page)

FixedSized NullTerminated NullStripped added, in Tunneling (see tutorial page)

RestreamData added, in Tunneling (see tutorial page)

Transformed added, in Tunneling (see tutorial page)

ProcessXor and ProcessRotateLeft added, in Tunneling (see tutorial page)

ExprAdapter Mapping Restreamed changed parameters order (decoders before encoders)

Adapter changed parameters, added *path* parameter to *_encode _decode _validate* methods (see tutorial page)

Lazy added, in Lazy equivalents category (see tutorial page)

LazyStruct LazyArray reimplemented with new lazy parsing semantics (see tutorial page)

LazySequence LazyRange LazyField(OnDemand) removed

LazyBound remains, but changed to parameter-less lambda (see tutorial page)

Probe Debugger updated, ProbeInto removed (see tutorial page)

Support classes

Container updated, uses *globalPrintFullStrings globalPrintFalseFlags globalPrintPrivateEntries*

Container updated, equality does not check hidden keys like _private or keys order

FlagsContainer removed

RestreamedBytesIO supports reading till EOF, enabling GreedyBytes GreedyString inside Bitwise Bytewise

HexString removed

Exceptions

FieldError was replaced with StreamError (raised when stream returns less than requested amount) and FormatFieldError (raised by FormatField class, for example if building Float from non-float value and struct.pack complains).

StreamError can be raised by most classes, when the stream is not seekable or tellable

StringError can be raised by classes like Bytes Const, when expected bytes but given unicode string as build value

BitIntegerError was replaced by IntegerError

Struct Sequence can raise IndexError KeyError when dictionaries are missing entries

RepeatError added

IndexFieldError added

CheckError added

NamedTupleError added

RawCopyError added

1.4 The Basics

1.4.1 Fields

Fields are the most fundamental unit of construction: they **parse** (read data from the stream and return an object) and **build** (take an object and write it down onto a stream). There are many kinds of fields, each working with a different type of data (numeric, boolean, strings, etc.).

Some examples of parsing:

```
>>> from construct import Int16ub, Int16ul
>>> Int16ub.parse(b"\x01\x02")
258
>>> Int16ul.parse(b"\x01\x02")
513
```

Some examples of building:

```
>>> from construct import Int16ub, Int16sb
>>> Int16ub.build(31337)
'zi'
>>> Int16sb.build(-31337)
'\x86\x97'
```

Other fields like:

```
>>> Flag.parse(b"\x01")
True
```

```
>>> Enum(Byte, g=8, h=11).parse(b"\x08")
'g'
>>> Enum(Byte, g=8, h=11).build(11)
b'\x0b'
```

```
>>> Float32b.build(12.345)
b'AE\x85\x1f'
>>> Single.parse(_)
12.345000267028809
```

1.4.2 Variable-length fields

```
>>> VarInt.build(1234567890)
b'\xd2\x85\xd8\xcc\x04'
>>> VarInt.sizeof()
construct.core.SizeofError: cannot calculate size
```

Fields are sometimes fixed size and some composites behave differently when they are composed of those. Keep that detail in mind. Classes that cannot determine size always raise SizeofError in response. There are few classes where same instance may return an integer or raise SizeofError depending on circumstances. Array size depends on whether count of elements is constant (can be a context lambda) and subcon is fixed size (can be variable size). For example, many classes take context lambdas and SizeofError is raised if the key is missing from the context.

```
>>> Int16ub[2].sizeof()
4
>>> VarInt[1].sizeof()
construct.core.SizeofError: cannot calculate size
```

1.4.3 Structs

For those of you familiar with C, Structs are very intuitive, but here's a short explanation for the larger audience. A Struct is a collection of ordered and usually named fields (field means an instance of Construct class), that are parsed/built in that same order. Names are used for two reasons: (1) when parsed, values are returned in a dictionary where keys are matching the names, and when build, each field gets build with a value taken from a dictionary from a matching key (2) fields parsed and built values are inserted into the context dictionary under matching names.

```
>>> format = Struct(
...     "signature" / Const(b"BMP"),
...     "width" / Int8ub,
...     "height" / Int8ub,
...     "pixels" / Array(this.width * this.height, Byte),
... )
>>> format.build(dict(width=3,height=2,pixels=[7,8,9,11,12,13]))
b'BMP\x03\x02\x07\x08\t\x0b\x0c\r'
>>> format.parse(b'BMP\x03\x02\x07\x08\t\x0b\x0c\r')
Container(signature=b'BMP', width=3, height=2, pixels=[7, 8, 9, 11, 12, 13])
```

Usually members are named but there are some classes that build from nothing and return nothing on parsing, so they have no need for a name (they can stay anonymous). Duplicated names within same struct can have unknown side effects.

```
>>> test = Struct(
...     Const(b"XYZ"),
...     Padding(2),
...     Pass,
...     Terminated,
... )
>>> test.build({})
b'XYZ\x00\x00'
>>> test.parse(_)
Container()
```

Note that this syntax works ONLY on CPython 3.6 (and PyPy any version) due to ordered keyword arguments. There is similar syntax for many other constructs.

```
>>> Struct(a=Byte, b=Byte, c=Byte, d=Byte)
```

Operator + can also be used to make Structs, and to merge them. Structs are embedded (not nested) when added.

```
>>> st = "count"/Byte + "items"/Byte[this.count] + Terminated
>>> st.parse(b"\x03\x01\x02\x03")
Container(count=3, items=[1, 2, 3])
```

Containers

What is that Container object, anyway? Well, a Container is a regular Python dictionary. It provides pretty-printing and allows accessing items as attributes as well as keys, and also preserves insertion order. Let's see more of those:

```
>>> st = Struct("float"/Single)
>>> x = st.parse(b"\x00\x00\x00\x01")
>>> x.float
1.401298464324817e-45
>>> x["float"]
1.401298464324817e-45
>>> repr(x)
Container(float=1.401298464324817e-45)
>>> print(x)
Container:
    float = 1.401298464324817e-45
```

As you can see, Containers provide human-readable representation of the data when printed, which is very important. By default, it truncates byte-strings and unicode-strings and hides EnumFlags unset flags (false values). If you would like a full print, you can use these functions:

```
>>> setGlobalPrintFalseFlags(True)
>>> setGlobalPrintFullStrings(True)
>>> setGlobalPrintPrivateEntries(True)
```

Thanks to blapid, containers can also be searched. Structs nested within Structs return containers within containers on parsing. One can search the entire "tree" of dicts for a particular name. Regular expressions are supported.

```
>>> con = Container(Container(a=1,d=Container(a=2)))
>>> con.search("a")
```

(continues on next page)

(continued from previous page)

```
1
>>> con.search_all("a")
[1, 2]
```

Nesting and embedding

Structs can be nested. Structs can contain other Structs, as well as any other constructs. Here's how it's done:

```
>>> st = Struct(
...     "inner" / Struct(
...         "data" / Bytes(4),
...     )
... )
>>> st.parse(b"1234")
Container(inner=Container(data=b'1234'))
>>> print(_)
Container:
    inner = Container:
        data = b'1234'
```

A Struct can be embedded into an enclosing Struct. This means that all fields of the embedded Struct get merged into the fields of the enclosing Struct. This is useful when you want to split a big Struct into multiple parts, and then combine them all into one Struct. If names are duplicated, consistency is not guaranteed (you should avoid that).

```
>>> outer = Struct(
...     Embedded(Struct(
...         "data" / Bytes(4),
...     )),
... )
>>> outer.parse(b"1234")
Container(data=b'1234')
```

Embedded structs should not be named, see *Embedded* .

Hidden context entries

There are few additional, hidden entries in the context. They are mostly used internally so they are not very well documented.

```
>>> d = Struct(
...     'x' / Computed(1),
...     'inner' / Struct(
...         'inner2' / Struct(
...             'x' / Computed(this._root.x),
...             'z' / Computed(this._params.z),
...             'zz' / Computed(this._root._.z),
...         ),
...     ),
...     Probe(),
... )
>>> setGlobalPrintPrivateEntries(True)
>>> d.parse(b'', z=2)
--------------------------------------------------
Probe, path is (parsing), into is None
```

(continues on next page)

(continued from previous page)

```
Container:
    _ = Container:
        z = 2
        _parsing = True
        _building = False
        _params = <recursion detected>
    _params = Container:
        z = 2
        _parsing = True
        _building = False
        _params = <recursion detected>
    _root = <recursion detected>
    _parsing = True
    _building = False
    _subcons = Container:
        x = <Renamed x +nonbuild <Computed +nonbuild>>
        inner = <Renamed inner +nonbuild <Struct +nonbuild>>
    _io = <_io.BytesIO object at 0x7fa29d7f9938>
    x = 1
    inner = Container:
        _io = <_io.BytesIO object at 0x7fa29d7f9938>
        inner2 = Container:
            _io = <_io.BytesIO object at 0x7fa29d7f9938>
            x = 1
            z = 2
            zz = 2
--------------------------------------------------
Container(x=1, inner=Container(inner2=Container(x=1, z=2, zz=2)))
```

Explanation as follows:

- _ means up-level in the context stack, every Struct does context nesting
- *_params* is the level on which externally provided values reside, those passed as parse() keyword arguments
- *_root* is the outer-most Struct, this entry might not exist if you do not use Structs
- *_parsing _building* are boolean values that are set by *parse build sizeof* public API methods
- *_subcons* is a list of Construct instances, this Struct members
- *_io* is a memory-stream or file-stream or whatever was provided to *parse_stream* public API method
- (parsed members are also added under matching names)

1.4.4 Sequences

Sequences are very similar to Structs, but operate with lists rather than containers. Sequences are less commonly used than Structs, but are very handy in certain situations. Since a list is returned in place of an attribute container, the names of the sub-constructs are not important. Two constructs with the same name will not override or replace each other. Names are used for the purposes of context dict.

```
>>> seq = Sequence(
...     Int16ub,
...     CString("utf8"),
...     GreedyBytes,
... )
```

Operator >> can also be used to make Sequences, or to merge them (but this syntax is not recommended).

```
>>> seq = Int16ub >> CString("utf8") >> GreedyBytes
>>> seq.parse(b"\x00\x80lalalaland\x00\x00\x00\x00\x00")
[128, 'lalalaland', b'\x00\x00\x00\x00']
```

1.4.5 Repeaters

Repeaters, as their name suggests, repeat a given unit for a specified number of times. At this point, we'll only cover static repeaters where count is a constant integer. Meta-repeaters take values at parse/build time from the context and they will be covered in the meta-constructs tutorial. Arrays and GreedyRanges differ from Sequences in that they are homogenous, they process elements of same kind. We have three kinds of repeaters.

Arrays have a fixed constant count of elements. Operator *[]* is used instead of calling the *Array* class (and is recommended syntax).

```
>>> d = Array(10, Byte) or Byte[10]
>>> d.parse(b"1234567890")
[49, 50, 51, 52, 53, 54, 55, 56, 57, 48]
>>> d.build([1,2,3,4,5,6,7,8,9,0])
b'\x01\x02\x03\x04\x05\x06\x07\x08\t\x00'
```

GreedyRange attempts to parse until EOF or subcon fails to parse correctly. Either way, when GreedyRange encounters either failure it seeks the stream back to a position after last successful subcon parsing. This means the stream must be seekable/tellable (doesnt work inside Bitwise).

```
>>> d = GreedyRange(Byte)
>>> d.parse(b"dsadhsaui")
[100, 115, 97, 100, 104, 115, 97, 117, 105]
```

RepeatUntil is different than the others. Each element is tested by a lambda predicate. The predicate signals when a given element is the terminal element. The repeater inserts all previous items along with the terminal one, and returns them as a list.

Note that all elements accumulated during parsing are provided as additional lambda parameter (second in order).

```
>>> d = RepeatUntil(lambda obj,lst,ctx: obj > 10, Byte)
>>> d.parse(b"\x01\x05\x08\xff\x01\x02\x03")
[1, 5, 8, 255]
>>> d.build(range(20))
b'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t\n\x0b'
```

```
>>> d = RepeatUntil(lambda x,lst,ctx: lst[-2:] == [0,0], Byte)
>>> d.parse(b"\x01\x00\x00\xff")
[1, 0, 0]
```

1.4.6 Processing on-the-fly

Data can be parsed and processed before further items get parsed. Hooks can be attached by using * operator.

Repeater classes like GreedyRange support indexing feature, which inserts incremental numbers into the context under *_index* key, in case you want to enumerate the objects. If you dont want to process further data, just raise CancelParsing from within the hook, and the parse method will exit clean.

```
def printobj(obj, ctx):
    print(obj)
    if ctx._index+1 >= 3:
        raise CancelParsing
st = Struct(
    "first" / Byte * printobj,
    "second" / Byte,
)
d = GreedyRange(st * printobj)
```

If you want to process gigabyte-sized data, then GreedyRange has an option to discard each element after it was parsed (and processed by the hook). Otherwise you would end up consuming gigabytes of RAM, because GreedyRange normally accumulates all parsed objects and returns them in a list.

```
d = GreedyRange(Struct(...) * printobj, discard=True)
```

1.5 The Basics, part 2

1.5.1 Integers and floats

Basic computer science 101. All integers follow the Int{8,16,24,32,64}{u,s}{b,l,n} and floats follow the Float{32,64}{b,l} naming patterns. Endianness can be either big-endian, little-endian or native. Integers can be signed or unsigned (non-negative only). Floats do not have a unsigned type.

```
>>> Int64sl.build(500)
b'\xf4\x01\x00\x00\x00\x00\x00\x00'
>>> Int64sl.build(-23)
b'\xe9\xff\xff\xff\xff\xff\xff\xff'
```

Few fields have aliases, Byte among integers and Single among floats.

```
Byte    <-->  Int8ub
Short   <-->  Int16ub
Int     <-->  Int32ub
Long    <-->  Int64ub
Single  <-->  Float32b
Double  <-->  Float64b
```

Integers can also be variable-length encoded for compactness. Google invented a popular encoding:

```
>>> VarInt.build(1234567890)
b'\xd2\x85\xd8\xcc\x04'
```

Long integers (or those of particularly odd sizes) can be encoded using a fixed-sized *BytesInteger*. Here is a 128-bit integer.

```
>>> BytesInteger(16).build(255)
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\xff'
```

Some numerical classes are implemented using *struct* module, others use BytesInteger field.

```
>>> FormatField("<","l").build(1)
b'\x01\x00\x00\x00'
```

(continues on next page)

(continued from previous page)

```
>>> BytesInteger(4, swapped=True).build(1)
b'\x01\x00\x00\x00'
```

1.5.2 Bytes and bits

Warning: Python 3 known problem:

Unprefixed string literals like "data" are on Python 3 interpreted as unicode. This casues failures when using fields like *Bytes*.

"Strings" of bytes (*str* in PY2 and *bytes* in PY3) can be moved around as-is. Bits are discussed in a later chapter.

```
>>> Bytes(5).build(b"12345")
b'12345'
>>> Bytes(5).parse(b"12345")
b'12345'
```

Bytes can also be consumed until end of stream. Tunneling is discussed in a later chapter.

```
>>> GreedyBytes.parse(b"39217839219...")
b'39217839219...'
```

1.5.3 Strings

Warning: Python 2 known problem:

Unprefixed string literals like "text" are on Python 2 interpreted as bytes. This causes failures when using fields that operate on unicode objects only like String* classes.

Note: Encodings like UTF8 UTF16 UTF32 (including little-endian) work fine with all String* classes. However two of them, PaddedString and CString, support only encodings listed explicitly in *possiblestringencodings*.

PaddedString is a fixed-length construct that pads built string with null bytes, and strips those same null bytes when parsing. Strings can also be trimmed when building. If you supply a too long string, the construct will chop it off apart instead of raising a StringError.

To be honest, using this class is not recommended. It is provided only for ancient data formats.

```
>>> PaddedString(10, "utf8").build("")
b'\xd0\x90\xd1\x84\xd0\xbe\xd0\xbd\x00\x00'
```

PascalString is a variable length string that is prefixed by a length field. This scheme was invented in Pascal language that put Byte field instead of C convention of appending null \0 byte at the end. Note that the length field does not need to be Byte, and can also be variable length itself, as shown below. VarInt is recommended when designing new protocols.

```
>>> PascalString(VarInt, "utf8").build("")
b'\x08\xd0\x90\xd1\x84\xd0\xbe\xd0\xbd'
```

CString is another string representation, that always ends with a null \0 terminating byte at the end. This scheme was invented in C language and is known in the computer science community very well. One of the authors, Kernighan or Ritchie, admitted that it was one of the most regretable design decisions in history.

```
>>> CString("utf8").build(b"hello")
b'hello\x00'
```

Last would be GreedyString which does the same thing as GreedyBytes, plus encoding. It reads until the end of stream and then decodes data using specified encoding. Greedy* classes are usually used with tunneling constructs, which are discussed in a later chapter.

```
>>> GreedyString("utf8").parse(b"329817392189")
'329817392189'
```

1.5.4 Mappings

Booleans are flags:

```
>>> Flag.parse(b"\x01")
True
>>> Flag.build(True)
b'\x01'
```

Enum translates between string labels and integer values. Parsing returns a string (if value has mapping) but returns an integer otherwise. This creates no problem since Enum can build from string and integer representations just the same. Note that resulting string has a special implementation, so it can be converted into a corresponding integer.

```
>>> d = Enum(Byte, one=1, two=2, four=4, eight=8)
>>> d.parse(b"\x01")
'one'
>>> int(d.parse(b"\x01"))
1
>>> d.parse(b"\xff")
255
>>> int(d.parse(b"\xff"))
255
```

Note that string values can also be obtained using attribute members.

```
>>> d.build(d.one or "one" or 1)
b'\x01'
>>> d.one
'one'
```

FlagsEnum decomposes an integer value into a set of string labels:

```
>>> d = FlagsEnum(Byte, one=1, two=2, four=4, eight=8)
>>> d.parse(b"\x03")
Container(one=True, two=True, four=False, eight=False)
>>> d.build(dict(one=True,two=True))
b'\x03'
```

Note that string values can also be obtained using attribute members.

```
>>> d.build(d.one|d.two or "one|two" or 1|2)
b'\x03'
```

Both Enum and FlagsEnum support merging labels from IntEnum and IntFlag (enum34 module):

```
import enum
class E(enum.IntEnum or enum.IntFlag):
    one = 1
    two = 2

Enum(Byte, E) <--> Enum(Byte, one=1, two=2)
FlagsEnum(Byte, E) <--> FlagsEnum(Byte, one=1, two=2)
```

For completeness, there is also Mapping class, but using it is not recommended. Consider it a last resort.

```
>>> x = object
>>> d = Mapping(Byte, {x:0})
>>> d.parse(b"\x00")
x
>>> d.build(x)
b'\x00'
```

1.5.5 Processing files

Warning: Python 3 known problem:

Opening a file without mode like `open(filename)` implies text mode, which cannot be parsed or build.

Constructs can parse both in-memory data (bytes) and binary files:

```
>>> d = Struct(...)
>>> d.parse(bytes(1000))
```

```
>>> with open('/dev/zero', 'rb') as f:
...     d.parse_stream(f)
```

```
>>> d.parse_file('/dev/zero')
```

1.5.6 Documenting fields

Top-most structures should have elaborate descriptions, documenting who made them and from what specifications. Individual fields can also have docstrings, but field names should be descriptive, not the docstrings.

```
"""
Full docstring with autor, email, links to RFC-alike pages.
""" * \
Struct(
    "title" / CString("utf8"),
    Padding(2) * "reserved, see 8.1",
)
```

1.6 The Bit/Byte Duality

1.6.1 History

In Construct 1.XX, parsing and building were performed at the bit level: the entire data was converted to a string of 1's and 0's, so you could really work with bit fields. Every construct worked with bits, except some (which were named ByteXXX) that worked on whole octets. This made it very easy to work with single bits, such as the flags of the TCP header, 7-bit ASCII characters, or fields that were not aligned to the byte boundary (nibbles et al).

This approach was easy and flexible, but had two main drawbacks:

- Most data is byte-aligned (with very few exceptions)
- The overhead was too big

Since constructs worked on bits, the data had to be first converted to a bit-string, which meant you had to hold the entire data set in memory. Not only that, but you actually held 8 times the size of the original data (it was a bit-string). According to some tests I made, you were limited to files of about 50MB (and that was slow due to page-thrashing).

So as of Construct 2.XX, all constructs work with bytes:

- Less memory consumption
- No unnecessary bytes-to-bits and bits-to-bytes coversions
- Can rely on python's built in struct module for numeric packing/unpacking (faster and tested)
- Can directly parse from and build to file-like objects (without in-memory buffering)

But how are we supposed to work with raw bits? The only difference is that we must explicitly declare that: certain fields like BitsInteger (Bit Nibble Octet are instances of BitsInteger) handle parsing and building of bit strings. There are also few fields like Struct and Flag that work with both byte-strings and bit-strings.

1.6.2 BitStruct

A BitStruct is a sequence of constructs that are parsed/built in the specified order, much like normal Structs. The difference is that BitStruct operates on bits rather than bytes. When parsing a BitStruct, the data is first converted to a bit stream (a stream of \x01 and \x00), and only then is it fed to the subconstructs. The subconstructs are expected to operate on bits instead of bytes. For reference look at the code below:

```
>>> d = BitStruct(
...     "a" / Flag,
...     "b" / Nibble,
...     "c" / BitsInteger(10),
...     "d" / Padding(1),
... )
>>> d.parse(b"\xbe\xef")
Container(a=True, b=7, c=887, d=None)
>>> d.sizeof()
2
```

BitStruct is actually just a wrapper for the *Bitwise* around a *Struct* .

1.6.3 Important notes

- BitStructs are non-nestable (because Bitwise are not nestable) so writing something like `BitStruct(BitStruct(Octet))` will not work. You can use regular Structs inside BitStructs.

- Byte aligned - The total size of the elements of a BitStruct must be a multiple of 8 (due to alignment issues). RestreamedBytesIO will raise an error if the amount of bits and bytes does not align properly.
- GreedyRange Pointer Lazy* - Do not place fields that do seeking/telling or lazy parsing inside bitwise, because RestreamedBytesIO offsets will turn out wrong, have unknown side-effects or raise unknown exceptions.
- Normal (byte-oriented) classes like Int* Float* can be used by wrapping in Bytewise. If you need to mix byte- and bit-oriented fields, you should use a BitStruct and Bytewise.
- Advanced classes like tunneling may not work in bitwise context. Only basic fields like integers were throughly tested.

1.6.4 Fields that work with bits

Those classes work exclusively in Bitwise context.

```
Bit    <--> BitsInteger(1)
Nibble <--> BitsInteger(4)
Octet  <--> BitsInteger(8)
```

1.6.5 Fields that work with bytes

Normal classes, that is those working with byte-streams, can be used on bit-streams by wrapping them with Bytewise. Its a wrapper that does the opposite of Bitwise, it transforms each 8 bits into 1 byte. The enclosing stream is a bit-stream but the subcon is provided a byte-stream.

```
>>> d = Bitwise(Struct(
...     'a' / Nibble,
...     'b' / Bytewise(Float32b),
...     'c' / Padding(4),
... ))
>>> d.parse(bytes(5))
Container(a=0, b=0.0, c=None)
>>> d.sizeof()
5
```

1.6.6 Fields that do both

Some simple fields (such as Flag Padding Pass Terminated) are ignorant to the granularity of the data they operate on. The actual granularity depends on the enclosing layers. Same applies to classes that are wrappers or adapters like Enum EnumFlags. Those classes do not care about granularity because they dont interact with the stream, its their subcons.

Here's a snippet of a code that operates on bytes:

```
>>> d = Struct(
...     Padding(2),
...     "x" / Flag,
...     Padding(5),
... )
>>> d.build(dict(x=5))
b'\x00\x00\x01\x00\x00\x00\x00\x00'
>>> d.sizeof()
8
```

And here's a snippet of a code that operates on bits. The only difference is BitStruct in place of a normal Struct:

```
>>> d = Bitwise(Struct(
...     Padding(2),
...     "x" / Flag,
...     Padding(5),
... ))
>>> d.build(dict(x=5))
b' '
>>> d.sizeof()
1
```

So unlike "classical Construct", there's no need for BytePadding and BitPadding. If Padding is enclosed by a BitStruct, it operates on bits, otherwise, it operates on bytes.

1.6.7 Fields that do not work and fail

Following classes may not work within Bitwise Bytewise depending one some circumstances. Actually this section applies to ByteSwapped BitsSwapped as well. Those 4 are macros and resolve to either Transformed or Restreamed depending if subcon is fixed-sized and therefore the data can be prefetched entirely. If yes, then it turns into Transformed and should work just fine, it not, then it turns into Restreamed which uses RestreamedBytesIO which has several limitations in its implementation. Milage may vary.

Those do use stream seeking or telling (or both):

- GreedyRange
- Union
- Select
- Padded (actually works)
- Aligned (actually works)
- Pointer
- Peek
- Seek
- Tell
- RawCopy
- Prefixed (actually works)
- PrefixedArray (actually works)
- NullTerminated (actually works unless consume=False)
- LazyStruct
- LazyArray

1.7 The Context

Meta constructs are the key to the declarative power of Construct. Meta constructs are constructs which are affected by the context of the construction (during parsing and building). The context is a dictionary that is created during the parsing and building process by Structs and Sequences, and is "propagated" down and up to all constructs along the

way, so that other members can access other members parsing or building resuslts. It basically represents a mirror image of the construction tree, as it is altered by the different constructs. Nested structs create nested contexts, just as they create nested containers.

In order to see the context, let's try this snippet:

```
>>> st = Struct(
...     "a" / Byte,
...     Probe(),
...     "b" / Byte,
...     Probe(),
... )
>>> st.parse(b"\x01\x02")
--------------------------------------------------
Container:
    a = 1
--------------------------------------------------
Container:
    a = 1
    b = 2
--------------------------------------------------
Container(a=1, b=2)
```

As you can see, the context looks different at different points of the construction.

You may wonder what does the little underscore ('_') that is found in the context means. It basically represents the parent node, like the '..' in unix pathnames ('../foo.txt'). We'll use it only when we refer to the context of upper layers.

Using the context is easy. All meta constructs take a function as a parameter, which is usually passed as a lambda function, although "big" named functions are just as good. This function, unless otherwise stated, takes a single parameter called ctx (short for context), and returns a result calculated from that context.

```
>>> st = Struct(
...     "count" / Byte,
...     "data" / Bytes(this.count),
... )
>>> st.parse(b"\x05abcde")
Container(count=5, data=b'abcde')
```

Of course a function can return anything (it does not need to depend on the context):

```
>>> Computed(lambda ctx: 7)
>>> Computed(lambda ctx: os.urandom(16))
```

1.7.1 Nesting and embedding

And here's how we use the special "_" name to get to the upper container in a nested containers situation (which happens when parsing nested Structs). Notice that *length1* is on different (upper) level than *length2*, therefore it exists within a different up-level containter.

```
>>> st = Struct(
...     "length1" / Byte,
...     "inner" / Struct(
...         "length2" / Byte,
...         "sum" / Computed(this._.length1 + this.length2),
...     ),
... )
```

(continues on next page)

(continued from previous page)

```
>>> st.parse(b"12")
Container(length1=49, inner=Container(length2=50, sum=99))
```

Context entries can also be passed directly through *parse* and *build* methods. However, one should take into account that some classes are nesting context (like Struct Sequence Union FocusedSeq LazyStruct), so entries passed to these end up on upper level. Compare examples:

```
>>> d = Bytes(this.n)
>>> d.parse(bytes(100), n=4)
b'\x00\x00\x00\x00'
```

```
>>> d = Struct(
...     "data" / Bytes(this._.n),
... )
>>> d.parse(bytes(100), n=4)
Container(data=b'\x00\x00\x00\x00')
```

Embedding also complicates using the context. Notice that *count* is on same level as *data* because embedding simply "levels the plainfield".

```
>>> outer = Struct(
...     "count" / Byte,
...     Embedded(Struct(
...         "data" / Bytes(this.count),
...     )),
... )
>>> outer.parse(b"\x041234")
Container(count=4, data=b'1234')
```

1.7.2 Refering to inlined constructs

If you need to refer to a subcon like Enum, that was inlined in the struct (and therefore wasnt assigned to any variable in the namespace), you can access it as Struct attribute under same name. This feature is particularly handy when using Enums and EnumFlags.

```
>>> d = Struct(
...     "animal" / Enum(Byte, giraffe=1),
... )
>>> d.animal.giraffe
'giraffe'
```

If you need to refer to the size of a field, that was inlined in the same struct (and therefore wasnt assigned to any variable in the namespace), you can use a special "_subcons" context entry that contains all Struct members. Note that you need to use a lambda (because *this* expression is not supported).

```
>>> d = Struct(
...     "count" / Byte,
...     "data" / Bytes(lambda this: this.count - this._subcons.count.sizeof()),
... )
>>> d.parse(b"\x05four")
Container(count=5)(data=b'four')
```

```
>>> d = Union(None,
...     "chars" / Byte[4],
...     "data" / Bytes(lambda this: this._subcons.chars.sizeof()),
... )
>>> d.parse(b"\x01\x02\x03\x04")
Container(chars=[1, 2, 3, 4], data=b'\x01\x02\x03\x04')
```

This feature is supported in same constructs as embedding: Struct Sequence FocusedSeq Union LazyStruct.

1.7.3 Using *this* expression

Certain classes take a number of elements, or something similar, and allow a callable to be provided instead. This callable is called at parsing and building, and is provided the current context object. Context is always a Container, not a dict, so it supports attribute as well as key access. Amazingly, this can get even more fancy. Tomer Filiba provided an even better syntax. The *this* singleton object can be used to build a lambda expression. All four examples below are equivalent, but first is recommended:

```
>>> this._.field
>>> lambda this: this._.field
>>> this["_"]["field"]
>>> lambda this: this["_"]["field"]
```

Of course, *this* expression can be mixed with other calculations. When evaluating, each instance of *this* is replaced by context Container which supports attribute access to keys.

```
>>> this.width * this.height - this.offset
```

When creating an Array ("items" field), rather than specifying a constant count, you can use a previous field value as count.

```
>>> st = Struct(
...     "count" / Rebuild(Byte, len_(this.items)),
...     "items" / Byte[this.count],
... )
>>> st.build(dict(items=[1,2,3,4,5]))
b'\x05\x01\x02\x03\x04\x05'
```

Switch can branch the construction path based on previously parsed value.

```
>>> st = Struct(
...     "type" / Enum(Byte, INT1=1, INT2=2, INT4=3, STRING=4),
...     "data" / Switch(this.type,
...     {
...         "INT1" : Int8ub,
...         "INT2" : Int16ub,
...         "INT4" : Int32ub,
...         "STRING" : String(10),
...     }),
... )
>>> st.parse(b"\x02\x00\xff")
Container(type='INT2', data=255)
>>> st.parse(b"\x04\abcdef\x00\x00\x00\x00")
Container(type='STRING', data=b'\x07bcdef')
```

1.7.4 Using *len_* expression

There used to be a bit of a hassle when you used built-in functions like *len sum min max abs* on context items. Built-in *len* takes a list and returns an integer but *len_* analog takes a lambda and returns a lambda. This allows you to use this kind of shorthand:

```
>>> len_(this.items)
>>> lambda this: len(this.items)
```

These can be used in newly added Rebuild wrappers that compute count/length fields from another list-alike field:

```
>>> st = Struct(
...     "count" / Rebuild(Byte, len_(this.items)),
...     "items" / Byte[this.count],
... )
>>> st.build(dict(items=[1,2,3,4,5]))
b'\x05\x01\x02\x03\x04\x05'
```

1.7.5 Using *obj_* expression

There is also an analog that takes (obj, context) or (obj, list, context) unlike *this* singleton which only takes a context (a single parameter):

```
>>> obj_ > 0
>>> lambda obj,ctx: obj > 0
```

These can be used in at least one construct:

```
>>> RepeatUntil(obj_ == 0, Byte).parse(b"aioweqnjkscs\x00")
[97, 105, 111, 119, 101, 113, 110, 106, 107, 115, 99, 115, 0]
```

1.7.6 Using *list_* expression

Warning: The *list_* expression is implemented but buggy, using it is not recommended at present time.

There is also a third expression that takes (obj, list, context) and computes on the second parameter (the list). In constructs that use lambdas with all 3 parameters, those constructs usually process lists of elements and the 2nd parameter is a list of elements processed so far.

These can be used in at least one construct:

```
>>> RepeatUntil(list_[-1] == 0, Byte).parse(b"aioweqnjkscs\x00")
[97, 105, 111, 119, 101, 113, 110, 106, 107, 115, 99, 115, 0]
```

In that example, *list_* gets substituted with following, at each iteration. Index -1 means last element:

```
list_ <- [97]
list_ <- [97, 105]
list_ <- [97, 105, 111]
list_ <- [97, 105, 111, 119]
...
```

1.7.7 Known deficiencies

Logical `and or not` operators cannot be used in this expressions. You have to either use a lambda or equivalent bitwise operators:

```
>>> ~this.flag1 | this.flag2 & this.flag3
>>> lambda this: not this.flag1 or this.flag2 and this.flag3
```

Contains operator `in` cannot be used in this expressions, you have to use a lambda:

```
>>> lambda this: this.value in (1, 2, 3)
```

Lambdas (unlike this expressions) are not compilable.

1.8 Miscellaneous

1.8.1 Special

Embedded

Special wrapper that allows outer multiple-subcons construct to merge fields from another multiple-subcons construct. Embedded does not change a field, only wraps it like a candy with a flag.

Warning: Can only be used between Struct Sequence FocusedSeq Union LazyStruct, although they can be used interchangably, for example Struct can embed fields from a Sequence. There is also *EmbeddedSwitch* macro that pseudo-embeds a Switch. Its not possible to embed IfThenElse.

```
>>> outer = Struct(
...     Embedded(Struct(
...         "data" / Bytes(4),
...     )),
... )
>>> outer.parse(b"1234")
Container(data=b'1234')
```

Renamed

Adds a name string to a field (which by default is None). This class is only used internally and you should use the / and * operators instead. Naming fields is needed when working with Structs and Unions, but also sometimes with Sequences and FocusedSeq.

```
"num" / Byte       <-->  Renamed(Byte, newname="num")
Byte * "comment"   <-->  Renamed(Byte, newdocs="comment")
Byte * parsedhook  <-->  Renamed(byte, newparsed=parsedhook)
```

1.8.2 Miscellaneous

Const

A constant value that is required to exist in the data and match a given value. If the value is not matching, ConstError is raised. Useful for so called magic numbers, signatures, asserting correct protocol version, etc.

```
>>> d = Const(b"IHDR")
>>> d.build(None)
b'IHDR'
>>> d.parse(b"JPEG")
construct.core.ConstError: expected b'IHDR' but parsed b'JPEG'
```

By default, Const uses a Bytes field with size matching the value. However, other fields can also be used:

```
>>> d = Const(255, Int32ul)
>>> d.build(None)
b'\xff\x00\x00\x00'
```

The shortcoming is that it only works if the amount and exact bytes are known in advance. To check if a "variable" data meets some criterium (not mere equality), you would need the *Check* class. There is also *OneOf* and *NoneOf* class.

Computed

Represents a value dynamically computed from the context. Computed does not read or write anything to the stream. It only computes a value (usually by extracting a key from a context dictionary) and returns its computed value as the result. Usually Computed fields are used for computations on the context dict. Context is explained in a previous chapter. However, Computed can also produce values based on external environment, random module, or constants. For example:

```
>>> st = Struct(
...     "width" / Byte,
...     "height" / Byte,
...     "total" / Computed(this.width * this.height),
... )
>>> st.parse(b"12")
Container(width=49, height=50, total=2450)
>>> st.build(dict(width=4,height=5))
b'\x04\x05'
```

```
>>> d = Computed(lambda ctx: os.urandom(10))
>>> d.parse(b"")
b'[\x86\xcc\xf1b\xd9\x10\x0f?\x1a'
```

Index

Fields that are inside Array GreedyRange RepeatUntil can reference their index within the outer list. This is being effectuated by repeater class maintaining a context entry "_index" and updating it between each iteration. Note that some classes do context nesting (like Struct), but they do copy the key over. You can access the key using Index class, or refer to the context entry directly, using *this._index* expression. Some constructions are only possible with direct method, when you want to use the index as parameter of a construct, like in *Bytes(this._index+1)*.

```
>>> d = Array(3, Index)
>>> d.parse(b"")
[0, 1, 2]
>>> d = Array(3, Struct("i" / Index))
>>> d.parse(b"")
[Container(i=0), Container(i=1), Container(i=2)]
```

```
>>> d = Array(3, Computed(this._index+1))
>>> d.parse(b"")
[1, 2, 3]
>>> d = Array(3, Struct("i" / Computed(this._index+1)))
>>> d.parse(b"")
[Container(i=1), Container(i=2), Container(i=3)]
```

Rebuild

When there is an array separated from its length field, the Rebuild wrapper can be used to measure the length of the list when building. Note that both the *len_* and *this* expressions are used as discussed in meta chapter. Only building is affected, parsing is simply deferred to subcon.

```
>>> st = Struct(
...     "count" / Rebuild(Byte, len_(this.items)),
...     "items" / Byte[this.count],
... )
>>> st.build(dict(items=[1,2,3]))
b'\x03\x01\x02\x03'
```

When the length field is directly before the items, *PrefixedArray* can be used instead:

```
>>> d = PrefixedArray(Byte, Byte)
>>> d.build([1,2,3])
b'\x03\x01\x02\x03'
```

Default

Allows to make a field have a default value, which comes handly when building a Struct from a dict with missing keys. Only building is affected, parsing is simply deferred to subcon.

```
>>> st = Struct(
...     "a" / Default(Byte, 0),
... )
>>> st.build(dict(a=1))
b'\x01'
>>> st.build(dict())
b'\x00'
```

Check

When fields are expected to be coherent in some way but integrity cannot be checked by merely comparing data with constant bytes using Const field, then a Check field can be put in place to get a key from context dict and check if the integrity is preserved. For example, maybe there is a count field (implied being non-negative but the field is signed type):

```
>>> st = Struct(
...     "num" / Int8sb,
...     "integrity1" / Check(this.num > 0),
... )
>>> st.parse(b"\xff")
ValidationError: check failed during parsing
```

Or there is a collection and a count provided and the count is expected to match the collection length (which might go out of sync by mistake). Note that Rebuild is more appropriate but the check is also possible:

```
>>> st = Struct(
...     "count" / Byte,
...     "items" / Byte[this.count],
... )
>>> st.build(dict(count=9090, items=[]))
FormatFieldError: packer '>B' error during building, given value 9090
>>> st = Struct(
...     "integrity" / Check(this.count == len_(this.items)),
...     "count" / Byte,
...     "items" / Byte[this.count],
... )
>>> st.build(dict(count=9090, items=[]))
ValidationError: check failed during building
```

Error

You can also explicitly raise an error, declaratively with a construct.

```
>>> Error.parse(b"")
ExplicitError: Error field was activated during parsing
```

FocusedSeq

When a sequence has some fields that could be ommited like Const Padding Terminated, the user can focus on one particular field that is useful. Only one field can be focused on, and can be referred by index or name. Other fields must be able to build without a value:

```
>>> d = FocusedSeq(1 or "num",
...     Const(b"MZ"),
...     "num" / Byte,
...     Terminated,
... )
>>> d.parse(b"MZ\xff")
255
>>> d.build(255)
b'MZ\xff'
```

Pickled

For convenience, arbitrary Python objects can be preserved using the famous pickle protocol. Almost any type can be pickled, but you have to understand that pickle uses its own (homebrew) protocol that is not a standard outside Python. Therefore, you can forget about parsing the binary blobs using other languages. There are also some minor

considerations, like pickle protocol requiring Python 3.0 version or so. Its useful, but it automates things beyond your understanding.

```
>>> obj = [1, 2.3, {}]
>>> Pickled.build(objobj)
b'\x80\x03]q\x00(K\x01G@\x02ffffff}q\x01e.'
>>> Pickled.parse(_)
[1, 2.3, {}]
```

Numpy

Numpy arrays can be preserved and retrived along with their element type (dtype), dimensions (shape) and items. This is effectuated using the Numpy binary protocol, so parsing blobs produced by this class with other langagues (or other frameworks than Numpy for that matter) is not possible. Otherwise you could use PrefixedArray but this class is more convenient.

```
>>> import numpy
>>> obj = numpy.asarray([1,2,3])
>>> Numpy.build(obj)
b"\x93NUMPY\x01\x00F\x00{'descr': '<i8', 'fortran_order': False, 'shape': (3,), }
↪
↪\n\x01\x00\x00\x00\x00\x00\x00\x00\x02\x00\x00\x00\x00\x00\x00\x00\x03\x00\x00\x00\x00\x00\x00\x00
↪"
```

NamedTuple

Both arrays, structs and sequences can be mapped to a namedtuple from collections module. To create a named tuple, you need to provide a name and a sequence of fields, either a string with space-separated names or a list of strings. Just like the stadard namedtuple does.

```
>>> d = NamedTuple("coord", "x y z", Byte[3])
>>> d = NamedTuple("coord", "x y z", Byte >> Byte >> Byte)
>>> d = NamedTuple("coord", "x y z", "x"/Byte + "y"/Byte + "z"/Byte)
>>> d.parse(b"123")
coord(x=49, y=50, z=51)
```

Timestamp

Datetimes can be represented using Timestamp class. It supports modern formats and even MSDOS one. Note however that this class is not guaranteed to provide "exact" accurate values, due to several reasons explained in the docstring.

```
>>> d = Timestamp(Int64ub, 1., 1970)
>>> d.parse(b'\x00\x00\x00\x00ZIz\x00')
<Arrow [2018-01-01T00:00:00+00:00]>
>>> d = Timestamp(Int32ub, "msdos", "msdos")
>>> d.parse(b'H9\x8c"')
<Arrow [2016-01-25T17:33:04+00:00]>
```

Hex and HexDump

Integers and bytes can be displayed in hex form, for convenience. Note that parsing still results in int-alike and bytes-alike objects, and those results are unmodified, the hex form appears only when pretty-printing. If you want to obtain

hexlified bytes, you need to use binascii.hexlify() on parsed results.

```
>>> d = Hex(Int32ub)
>>> obj = d.parse(b"\x00\x00\x01\x02")
>>> obj
258
>>> print(obj)
0x00000102
```

```
>>> d = Hex(GreedyBytes)
>>> obj = d.parse(b"\x00\x00\x01\x02")
>>> obj
b'\x00\x00\x01\x02'
>>> print(obj)
unhexlify('00000102')
```

```
>>> d = Hex(RawCopy(Int32ub))
>>> obj = d.parse(b"\x00\x00\x01\x02")
>>> obj
{'data': b'\x00\x00\x01\x02',
 'length': 4,
 'offset1': 0,
 'offset2': 4,
 'value': 258}
>>> print(obj)
unhexlify('00000102')
```

Another variant is hexdumping, which shows both ascii representaion, hexadecimal representation, and offsets. Functionality is identical.

```
>>> d = HexDump(GreedyBytes)
>>> obj = d.parse(b"\x00\x00\x01\x02")
>>> obj
b'\x00\x00\x01\x02'
>>> print(obj)
hexundump('''
0000   00 00 01 02                                       ....
''')
```

```
>>> d = HexDump(RawCopy(Int32ub))
>>> obj = d.parse(b"\x00\x00\x01\x02")
>>> obj
{'data': b'\x00\x00\x01\x02',
 'length': 4,
 'offset1': 0,
 'offset2': 4,
 'value': 258}
>>> print(obj)
hexundump('''
0000   00 00 01 02                                       ....
''')
```

1.8.3 Conditional

Union

Treats the same data as multiple constructs (similar to C union statement) so you can "look" at the data in multiple views.

When parsing, all fields read the same data bytes, but stream remains at initial offset (or rather seeks back to original position after each subcon was parsed), unless parsefrom selects a subcon by index or name. When building, the first subcon that can find an entry in the dict (or builds from None, so it does not require an entry) is automatically selected.

Warning: If you skip *parsefrom* parameter then stream will be left back at starting offset, not seeked to any common denominator.

```
>>> d = Union(0,
...     "raw" / Bytes(8),
...     "ints" / Int32ub[2],
...     "shorts" / Int16ub[4],
...     "chars" / Byte[8],
... )
>>> d.parse(b"12345678")
Container(raw=b'12345678', ints=[825373492, 892745528], shorts=[12594, 13108, 13622,␣
↪14136], chars=[49, 50, 51, 52, 53, 54, 55, 56])
>>> d.build(dict(chars=range(8)))
b'\x00\x01\x02\x03\x04\x05\x06\x07'
```

```
Note that this syntax works ONLY on CPython 3.6 (and PyPy any version) due to ordered␣
↪keyword arguments. There is similar syntax for many other constructs.
>>> Union(0, raw=Bytes(8), ints=Int32ub[2], shorts=Int16ub[4], chars=Byte[8])
```

Select

Attempts to parse or build each of the subcons, in order they were provided.

```
>>> d = Select(Int32ub, CString("utf8"))
>>> d.build(1)
b'\x00\x00\x00\x01'
>>> d.build(u"")
b'\xd0\x90\xd1\x84\xd0\xbe\xd0\xbd\x00'
```

```
Note that this syntax works ONLY on CPython 3.6 (and PyPy any version) due to ordered␣
↪keyword arguments. There is similar syntax for many other constructs.
>>> Select(num=Int32ub, text=CString("utf8"))
```

Optional

Attempts to parse or build the subconstruct. If it fails during parsing, returns a None. If it fails during building, it puts nothing into the stream.

```
>>> d = Optional(Int64ul)
>>> d.parse(b"12345678")
```

(continues on next page)

(continued from previous page)

```
4050765991979987505
>>> d.parse(b"")
None
```

```
>>> d.build(1)
b'\x01\x00\x00\x00\x00\x00\x00\x00'
>>> d.build(None)
b''
```

If

Parses or builds the subconstruct only if a certain condition is met. Otherwise, returns a None when parsing and puts nothing when building. The condition is a lambda that computes on the context just like in Computed examples.

```
>>> d = If(this.x > 0, Byte)
>>> d.build(255, x=1)
b'\xff'
>>> d.build(255, x=0)
b''
```

IfThenElse

Branches the construction path based on a given condition. If the condition is met, the `thensubcon` is used, otherwise the `elsesubcon` is used. Fields like Pass and Error can be used here. Just for your curiosity, If is just a macro around this class.

```
>>> d = IfThenElse(this.x > 0, VarInt, Byte)
>>> d.build(255, x=1)
b'\xff\x01'
>>> d.build(255, x=0)
b'\xff'
```

In particular, you can use different subcons for parsing and building. The context entries have boolean values and always exist (sizeof has both values as False). For convenience, those two entries are duplicated in Struct Sequence FocusedSeq Union nested contexts. You dont need to reach for the top-most entry. This comes handy when using hackish constructs to achieve some complex semantics that are not available in the core library.

```
Struct(
    If(this._parsing, ...),
    If(this._building, ...),
)
```

Switch

Branches the construction based on a return value from a context function. This is a more general implementation than IfThenElse. If no cases match the actual, it just passes successfully, although that behavior can be overriden.

```
>>> d = Switch(this.n, { 1:Int8ub, 2:Int16ub, 4:Int32ub })
>>> d.build(5, n=1)
b'\x05'
>>> d.build(5, n=4)
b'\x00\x00\x00\x05'
```

```
>>> d = Switch(this.n, {}, default=Byte)
>>> d.parse(b"\x01", n=255)
1
>>> d.build(1, n=255)
b"\x01"
```

EmbeddedSwitch

Macro that simulates embedding Switch, which under new embedding semantics is not possible. This macro does NOT produce a Switch. It generates classes that behave the same way as you would expect from embedded Switch, only that. Instance created by this macro CAN be embedded.

All fields should have unique names. Otherwise fields that were not selected during parsing may return None and override other fields context entries that have same name. This is because *If* field returns None value if condition is not met, but the Struct inserts that None value into the context entry regardless.

```
d = EmbeddedSwitch(
    Struct(
        "type" / Byte,
    ),
    this.type,
    {
        0: Struct("name" / PascalString(Byte, "utf8")),
        1: Struct("value" / Byte),
    }
)

# generates essentially following
d = Struct(
    "type" / Byte,
    "name" / If(this.type == 0, PascalString(Byte, "utf8")),
    "value" / If(this.type == 1, Byte),
)

# both parse like following
>>> d.parse(b"\x00\x00")
Container(type=0, name=u'', value=None)
>>> d.parse(b"\x01\x00")
Container(type=1, name=None, value=0)
```

StopIf

Checks for a condition after each element, and stops a Struct Sequence GreedyRange from parsing or building following elements.

```
Struct('x'/Byte, StopIf(this.x == 0), 'y'/Byte)
Sequence('x'/Byte, StopIf(this.x == 0), 'y'/Byte)
GreedyRange(FocusedSeq(0, 'x'/Byte, StopIf(this.x == 0)))
```

1.8.4 Alignment and padding

Padding

Adds additional null bytes (a filler) analog to Padded but without a subcon that follows it. This field is usually anonymous inside a Struct. Internally this is just Padded(Pass).

```
>>> d = Padding(4) or Padded(4, Pass)
>>> d.parse(b"****")
None
>>> d.build(None)
b'\x00\x00\x00\x00'
```

Padded

Appends additional null bytes after subcon to achieve a fixed length. Note that implementation of this class uses stream.tell() to find how many bytes were written by the subcon.

```
>>> d = Padded(4, Byte)
>>> d.build(255)
b'\xff\x00\x00\x00'
```

Similar effect can be obtained using FixedSized, but the implementation is rather different. FixedSized uses a separate BytesIO, which means that Greedy* fields should work properly with it (and fail with Padded) and also the stream does not need to be tellable (like pipes sockets etc).

Aligned

Appends additional null bytes after subcon to achieve a given modulus boundary. This implementation also uses stream.tell().

```
>>> d = Aligned(4, Int16ub)
>>> d.build(1)
b'\x00\x01\x00\x00'
```

AlignedStruct

Automatically aligns each member to modulus boundary. It does NOT align entire Struct, but each member separately.

```
>>> d = AlignedStruct(4, "a"/Int8ub, "b"/Int16ub)
>>> d.build(dict(a=0xFF,b=0xFFFF))
b'\xff\x00\x00\x00\xff\xff\x00\x00'
```

1.9 Stream manipulation

Note: Certain constructs are available only for seekable or tellable streams (in-memory and files). Sockets and pipes do not support neither, so you'll have to first read the data from the stream and parse it in-memory, or use experimental *Rebuffered* wrapper.

1.9.1 Field wrappers

Pointer allows for non-sequential construction. The pointer first moves the stream into new position, does the construction, and then restores the stream back to original position. This allows for random-access within the stream.

```
>>> d = Pointer(8, Bytes(1))
>>> d.parse(b"abcdefghijkl")
b'i'
>>> d.build(b"Z")
b'\x00\x00\x00\x00\x00\x00\x00\x00Z'
```

Peek parses a field but restores the stream position afterwards (it does peeking). Building does nothing, it does NOT defer to subcon.

```
>>> d = Sequence(Peek(Int16ul), Peek(Int16ub))
>>> d.parse(b"\x01\x02")
[513, 258]
>>> d.sizeof()
0
```

1.9.2 Pure side effects

Seek makes a jump within the stream and leaves it there, for other constructs to follow up from that location. It does not read or write anything to the stream by itself.

```
>>> d = Sequence(Bytes(10), Seek(5), Byte)
>>> d.build([b"0123456789", None, 255])
b'01234\xff6789'
```

Tell checks the current stream position and returns it. The returned value gets automatically inserted into the context dictionary. It also does not read or write anything to the stream by itself.

```
>>> d = Struct("num"/VarInt, "offset"/Tell)
>>> d.parse(b"X")
Container(num=88, offset=1)
>>> d.build(dict(num=88))
b'X'
```

1.9.3 Other fields

Pass literally does nothing. It is mostly used internally by If(IfThenElse) and Padding(Padded).

```
>>> Pass.parse(b"")
None
>>> Pass.build(None)
b''
>>> Pass.sizeof()
0
```

Terminated only works during parsing. It checks if the stream reached EOF and raises error if not.

```
>>> Terminated.parse(b"")
None
>>> Terminated.parse(b"remaining")
construct.core.TerminatedError: expected end of stream
```

1.10 Tunneling tactics

1.10.1 Obtaining raw bytes

When some value needs to be processed as both a parsed object and its raw bytes representation, both of these can be obtained using RawCopy. You can build from either the object or raw bytes as well. Dict also happen to contain the stream offsets, if you need to know at which position it resides in the stream or if you need to know its size in bytes.

```
>>> d = RawCopy(Byte)
>>> d.parse(b"\xff")
Container(data=b'\xff', value=255, offset1=0, offset2=1, length=1)
>>> d.build(dict(data=b"\xff"))
b'\xff'
>>> d.build(dict(value=255))
b'\xff'
```

1.10.2 Endianness

When little endianness is needed, either use integer fields like *Int*l* or *BytesInteger(swapped=True)* or swap bytes of an arbitrary field:

```
Int24ul <--> ByteSwapped(Int24ub) <--> BytesInteger(3, swapped=True) <-->␣
↪ByteSwapped(BytesInteger(3))
```

```
>>> Int24ul.build(0x010203)
b'\x03\x02\x01'
```

When bits within each byte need to be swapped, there is another wrapper:

```
>>> d = Bitwise(Bytes(8))
>>> d.parse(b"\x01")
b'\x00\x00\x00\x00\x00\x00\x00\x01'
>>> d = BitsSwapped(Bitwise(Bytes(8)))
>>> d.parse(b"\x01")
b'\x01\x00\x00\x00\x00\x00\x00\x00'
```

1.10.3 Working with bytes subsets

Greedy* constructs consume as much data as possible. This is convenient when building from a list of unknown length but becomes a problem when parsing it back and the list needs to be separated from following data. This can be achieved either by prepending a byte count (see Prefixed) or by prepending an element count (see PrefixedArray):

VarInt encoding is recommended because it is both compact and never overflows. Also and optionally, the length field can include its own size. If so, length field must be of fixed size.

```
>>> d = Prefixed(VarInt, GreedyRange(Int32ul))
>>> d.parse(b"\x08abcdefgh")
[1684234849, 1751606885]
```

```
>>> d = PrefixedArray(VarInt, Int32ul)
>>> d.parse(b"\x02abcdefgh")
[1684234849, 1751606885]
```

There are also other means of restricting constructs to substreamed data. All 3 classes below work by substreaming data, meaning the subcon is not given the original stream but a new BytesIO made out of pre-read bytes. This allows Greedy* fields to work properly.

FixedSized consumes a specified amount and then exposes inner construct to a new stream build out of those bytes. When building, it appends a padding to make a specified total.

```
>>> d = FixedSized(10, Byte)
>>> d.parse(b'\xff\x00\x00\x00\x00\x00\x00\x00\x00\x00')
255
```

FixedSized is similar to Padded. The results seem identical but the implementation is entirely different. FixedSized uses a substream and Padded uses stream.tell(). Therefore:

```
# valid
FixedSized(10, GreedyBytes)
# UNSAFE
Padded(10, GreedyBytes)
```

NullTerminated consumes bytes up to first occurance of the term. When building, it just writes the subcon followed by the term.

```
>>> d = NullTerminated(Byte)
>>> d.parse(b'\xff\x00')
255
```

NullStripped consumes bytes till EOF, and for that matter should be restricted by Prefixed FixedSized etc, and then strips paddings. Subcon is parsed using a new stream build using those stripped bytes. When building, it just builds the subcon as-is.

```
>>> d = NullStripped(Byte)
>>> d.parse(b'\xff\x00\x00')
255
```

1.10.4 Working with different bytes

RestreamData allows you to insert a field that parses some data that came either from some other field, from the context (like Bytes) or some literal hardcoded value in your code. Comes handy when for example, you are testing a large struct by parsing null bytes, but some field is unable to parse null bytes (like Numpy). It substitutes the stream with another data for the purposes of parsing a particular field in a Struct.

Instead of data itself (bytes object) you can reference another stream (taken from the context like *this._stream*) or use a Construct that parses into bytes (including those exposed via context like *this._subcons.field*).

```
>>> d = RestreamData(b"\x01", Int8ub)
>>> d.parse(b"")
1
>>> d.build(0)
b''
```

```
>>> d = RestreamData(NullTerminated(GreedyBytes), Int16ub)
>>> d.parse(b"\x01\x02\x00")
0x0102

>>> d = RestreamData(FixedSized(2, GreedyBytes), Int16ub)
```

(continues on next page)

(continued from previous page)

```
>>> d.parse(b"\x01\x02\x00")
0x0102
```

```
d = Struct(
    "numpy_data" / Computed(b"\x93NUMPY\x01\x00F\x00{'descr': '<i8', 'fortran_order': False, 'shape': (3,), }
↪\n\x01\x00\x00\x00\x00\x00\x00\x00\x02\x00\x00\x00\x00\x00\x00\x00\x03\x00\x00\x00\x00\x00\x00\x00
↪"),
    "numpy1" / RestreamData(this.numpy_data, Numpy),
    "numpy2" / Numpy, # this would fail when parsing null bytes
)
d.parse(bytes(1000))
```

Transformed allows you to process data before it gets into subcon (and after data left it) using simple bytes-to-bytes transformations. In fact, all core classes (like Bitwise) that use Restreamed also use Transformed. The only difference is that Transformed prefetches all bytes and transforms them in advance, but Restreamed fetches a unit at a time (few bytes usually). Therefore Restreamed can handle variable-sized fields, while Transformed works only with fixed-sized fields. For example:

```
>>> d = Transformed(Bytes(16), bytes2bits, 2, bits2bytes, 2)
>>> d.parse(b"\x00\x00")
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
```

Transformed can also process unknown amount of bytes, if that amount is entire data. Decode amount and encode amount are then set to None.

```
>>> d = Transformed(GreedyBytes, bytes2bits, None, bits2bytes, None)
>>> d.parse(b"\x00\x00")
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
```

```
# Bitwise implementation
try:
    size = subcon.sizeof()
    macro = Transformed(subcon, bytes2bits, size//8, bits2bytes, size//8)
except SizeofError:
    macro = Restreamed(subcon, bytes2bits, 1, bits2bytes, 8, lambda n: n//8)
```

Restreamed is similar to Transformed, but the main difference is that Transformed requires fixed-sized subcon because it reads all bytes in advance, processes them, and then feeds them to the subcon. Restreamed on the other hand, reads few bytes at a time, the minimum amount on each stream read. Since both are used mostly internally, there is no tutorial how to use it, other than this short code above.

1.10.5 Processing data with XOR and ROL

This chapter is mostly relevant to KaitaiStruct compiler implementation, as following constructs exist mostly for that purpose.

Data can be transformed by XORing with a single or several bytes, and the key can also be taken from the context at runtime. Key can be of any positive length.

```
>>> d = ProcessXor(0xf0 or b'\xf0', Int16ub)
>>> d.parse(b"\x00\xff")
0xf00f
```

(continues on next page)

(continued from previous page)

```
>>> d.sizeof()
2
```

Data can also be rotated (cycle shifted). Rotation is to the left on parsing, and to the right on building. Amount is in bits, and can be negative to make rotation right instead of left. Group size defines the size of chunks to which rotation is applied.

```
>>> d = ProcessRotateLeft(4, 1, Int16ub)
>>> d.parse(b'\x0f\xf0')
0xf00f
>>> d = ProcessRotateLeft(4, 2, Int16ub)
>>> d.parse(b'\x0f\xf0')
0xff00
>>> d.sizeof()
2
```

Note that the classes read entire stream till EOF so they should be wrapped in FixedSized Prefixed etc unless you actually want to process the entire remaining stream.

1.10.6 Compression and checksuming

Data can be easily checksummed. Note that checksum field does not need to be Bytes, and lambda may return an integer or otherwise.

```
import hashlib
d = Struct(
    "fields" / RawCopy(Struct(
        Padding(1000),
    )),
    "checksum" / Checksum(Bytes(64),
        lambda data: hashlib.sha512(data).digest(),
        this.fields.data),
)
d.build(dict(fields=dict(value={})))
```

```
import hashlib
d = Struct(
    "offset" / Tell,
    "checksum" / Padding(64),
    "fields" / RawCopy(Struct(
        Padding(1000),
    )),
    "checksum" / Pointer(this.offset, Checksum(Bytes(64),
        lambda data: hashlib.sha512(data).digest(),
        this.fields.data)),
)
d.build(dict(fields=dict(value={})))
```

Data can also be easily compressed. Supported encodings include zlib/gzip/bzip2/lzma and entire codecs module. When parsing, entire stream is consumed. When building, puts compressed bytes without marking the end. This construct should be used with *Prefixed* or entire stream.

```
>>> d = Prefixed(VarInt, Compressed(GreedyBytes, "zlib"))
>>> d.build(bytes(100))
b'\x0cx\x9cc`\xa0=\x00\x00\x00d\x00\x01'
```

1.11 Lazy parsing

1.11.1 Lazy

This wrapper allows you to do lazy parsing of individual fields inside a normal Struct (without using LazyStruct which may not in every scenario). It is also used by KaitaiStruct compiler to emit *instances* because those are not processed greedily, and they may refer to other not yet parsed fields. Those are 2 entirely different applications but semantics are the same.

```
>>> d = Lazy(Byte)
>>> x = d.parse(b'\x00')
>>> x
<function construct.core.Lazy._parse.<locals>.execute>
>>> x()
0
>>> d.build(0)
b'\x00'
>>> d.build(x)
b'\x00'
>>> d.sizeof()
1
```

1.11.2 LazyStruct

Equivalent to *Struct*, but when this class is parsed, most fields are not parsed (they are skipped if their size can be measured by _actualsize or _sizeof method). See its docstring for details.

Fields are parsed depending on some factors:

- Some fields like Int* Float* Bytes(5) Array(5,Byte) Pointer are fixed-size and are therefore skipped. Stream is not read.
- Some fields like Bytes(this.field) are variable-size but their size is known during parsing when there is a corresponding context entry. Those fields are also skipped. Stream is not read.
- Some fields like Prefixed PrefixedArray PascalString are variable-size but their size can be computed by partially reading the stream. Only first few bytes are read (the lengthfield).
- Other fields like VarInt need to be parsed. Stream position that is left after the field was parsed is used.
- Some fields may not work properly, due to the fact that this class attempts to skip fields, and parses them only out of necessity. Miscellaneous fields often have size defined as 0, and fixed sized fields are skippable.

Note there are restrictions:

- If a field like Bytes(this.field) references another field in the same struct, you need to access the referenced field first (to trigger its parsing) and then you can access the Bytes field. Otherwise it would fail due to missing context entry.
- If a field references another field within inner (nested) or outer (super) struct, things may break. Context is nested, but this class was not rigorously tested in that manner.

Building and sizeof are greedy, like in Struct.

1.11.3 LazyArray

Equivalent to *Array*, but the subcon is not parsed when possible (it gets skipped if the size can be measured by _actualsize or _sizeof method). See its docstring for details. The restrictions are identical as in LazyStruct.

1.11.4 LazyBound

Field that binds to the subcon only at runtime (during parsing and building, not ctor). Useful for recursive data structures, like linked-lists and trees, where a construct needs to refer to itself (while it does not exist yet in the namespace).

Note that it is possible to obtain same effect without using this class, using a loop. However there are usecases where that is not possible (if remaining nodes cannot be sized-up, and there is data following the recursive structure). There is also a significant difference, namely that LazyBound actually does greedy parsing while the loop does lazy parsing. See examples.

To break recursion, use *If* field. See examples.

```
d = Struct(
    "value" / Byte,
    "next" / If(this.value > 0, LazyBound(lambda: d)),
)

>>> print(d.parse(b"\x05\x09\x00"))
Container:
    value = 5
    next = Container:
        value = 9
        next = Container:
            value = 0
            next = None
```

```
d = Struct(
    "value" / Byte,
    "next" / GreedyBytes,
)

data = b"\x05\x09\x00"
while data:
    x = d.parse(data)
    data = x.next
    print(x)

# print outputs
Container:
    value = 5
    next = \t\x00 (total 2)
# print outputs
Container:
    value = 9
    next = \x00 (total 1)
# print outputs
Container:
    value = 0
    next =  (total 0)
```

1.12 Adapters and Validators

1.12.1 Adapting

Adapting is the process of converting one representation of an object to another. One representation is usually "lower" (closer to the byte level), and the other "higher" (closer to the python object model). The process of converting the lower representation to the higher one is called decoding, and the process of converting the higher level representation to the lower one is called encoding. Encoding and decoding are expected to be symmetrical, so that they counter-act each other `encode(decode(x)) == x` and `decode(encode(x)) == x`.

Custom adapter classes derive from the abstract Adapter class, and implement their own versions of *_decode* and *_encode*, as shown below:

```
class IpAddressAdapter(Adapter):
    def _decode(self, obj, context, path):
        return ".".join(map(str, obj))

    def _encode(self, obj, context, path):
        return list(map(int, obj.split(".")))

IpAddress = IpAddressAdapter(Byte[4])
```

As you can see, the IpAddressAdapter encodes a string of the format "XXX.XXX.XXX.XXX" to a list of 4 integers like [XXX, XXX, XXX, XXX]. This representation then gets handed over to Array(4, Byte) which turns it into bytes.

Note that the adapter does not perform any manipulation of the stream, it only converts between objects!

```
class Adapter(Subconstruct):
    def _parse(self, stream, context, path):
        return self._decode(self.subcon._parse(stream, context, path), context, path)

    def _build(self, obj, stream, context, path):
        return self.subcon._build(self._encode(obj, context, path), stream, context,␣
↪path)
```

This is called separation of concern, and is a key feature of component-oriented programming. It allows us to keep each component very simple and unaware of its consumers. Whenever we need a different representation of the data, we don't need to write a new Construct – we only write the suitable adapter.

So, let's see our adapter in action:

```
>>> IpAddress.parse(b"\x01\x02\x03\x04")
'1.2.3.4'
>>> IpAddress.build("192.168.2.3")
b'\xc0\xa8\x02\x03'
```

Having the representation separated from the actual parsing or building means an adapter is loosely coupled with its underlying construct. As with enums for example, we can use the same enum for *Byte* or *Int32sl* or *VarInt*, as long as the underlying construct returns an object that we can map. Moreover, we can stack several adapters on top of one another, to create a nested adapter.

Using expressions instead of classes

Adaters can be created declaratively using ExprAdapter. Note that this construction is not recommended, unless its much cleaner than Adapter. Use can use *obj_* expression to generate lambdas that operate on the object passed around.

For example, month in object model might be *1..12* but data format saves it as *0..11*.

```
>>> d = ExprAdapter(Byte, obj_+1, obj_-1)
>>> d.parse(b'\x04')
5
>>> d.build(5)
b'\x04'
```

Or another example, where some of the bits are unset in both parsed/build objects:

```
>>> d = ExprSymmetricAdapter(Byte, obj_ & 0b00001111)
>>> d.parse(b"\xff")
15
>>> d.build(255)
b'\x0f'
```

1.12.2 Validating and filtering

Validating means making sure the parsed/built object meets a given condition. Validators simply raise *ValidationError* if the lambda predicate indicates False when called with the actual object. You can write custom validators by deriving from the Validator class and implementing the *_validate* method.

```
class VersionNumberValidator(Validator):
    def _validate(self, obj, context, path):
        return obj in [1,2,3]

VersionNumber = VersionNumberValidator(Byte)
```

```
>>> VersionNumber.build(3)
b'\x03'
>>> VersionNumber.build(88)
ValidationError: object failed validation: 88
```

For reference, this is how it works under the hood (in core library):

```
class Validator(SymmetricAdapter):
    def _decode(self, obj, context, path):
        if not self._validate(obj, context, path):
            raise ValidationError("object failed validation: %s" % (obj,))
        return obj
```

Using expressions instead of classes

Validators can also be created declaratively using ExprValidator. Unfortunately *obj_* expression does not work with *in* (contains) operator, nor with *and or not* logical operators. But it still has the advantage that it can be declared inlined. Adapter and Validator derived classes cannot be inlined inside a Struct.

For example, if 7 out of 8 bits are not allowed to be set (like a flag boolean):

```
>>> d = ExprValidator(Byte, obj_ & 0b11111110 == 0)
>>> d.build(1)
b'\x01'
>>> d.build(88)
ValidationError: object failed validation: 88
```

1.13 Extending Construct

1.13.1 Adapters

Adapters are the standard way to extend and customize the library. Adapters operate at the object level (unlike constructs, which operate at the stream level), and are thus easy to write and more flexible. For more info see the adapter tutorial.

In order to write custom adapters, implement `_decode` and `_encode`:

```
class MyAdapter(Adapter):
    def _decode(self, obj, context, path):
        # called at parsing time to return a modified version of obj
        pass

    def _encode(self, obj, context, path):
        # called at building time to return a modified version of obj
        pass
```

1.13.2 Constructs

Generally speaking, you should not write constructs by yourself:

- It's a craft that requires skills and understanding of the internals of the library (which change over time).
- Adapters should really be all you need and are much simpler to implement.
- To make things faster, try using compilation feature, or pypy. The python-level classes are as fast as it gets, assuming generality.

The only reason you might want to write a custom class is to achieve something that's not currently possible. This might be a construct that computes/corrects the checksum of data, altough that already exists. Or a compression, or hashing. These also exist. But surely there is something that was not invented yet. If you need a semantics modification to an existing class, you can post a feature request, or copy the code of existing class into your project and modify it.

There are at least two kinds of constructs: raw construct and subconstructs.

Raw constructs

Deriving directly from class `Construct`, raw constructs can do as they wish by implementing `_parse`, `_build`, and `_sizeof`:

```
class MyConstruct(Construct):
    def _parse(self, stream, context, path):
        # read from the stream
        # return object
        pass

    def _build(self, obj, stream, context, path):
        # write obj to the stream
        # return same value (obj) or a modified value
        # that will replace the context dictionary entry
        pass

    def _sizeof(self, context, path):
```

(continues on next page)

(continued from previous page)

```
        # return computed size (when fixed size or depends on context)
        # or raise SizeofError (when variable size or unknown)
        pass
```

Variable size fields typically raise `SizeofError`, for example VarInt and CString.

Subconstructs

Deriving from class Subconstruct, these wrap an inner construct, inheriting it's properties (name and flags). In their `_parse` and `_build` and `_sizeof` methods, they will call `self.subcon._parse` and `self.subcon._build` and `self.subcon._sizeof` respectively.

```
class MySubconstruct(Subconstruct):
    def __init__(self, subcon):
        self.name = subcon.name
        self.subcon = subcon
        self.flagbuildnone = subcon.flagbuildnone
        self.flagembedded = subcon.flagembedded

    def _parse(self, stream, context, path):
        obj = self.subcon._parse(stream, context, path)
        # do something with obj
        return obj

    def _build(self, obj, stream, context, path):
        # do something with obj
        return self.subcon._build(obj, stream, context, path)
        # return same value (obj) or a modified value
        # that will replace the context dictionary entry

    def _sizeof(self, context, path):
        # if not overriden, defers to subcon size
        return self.subcon._sizeof(context, path)
```

1.14 Debugging Construct

Programming data structures in Construct is much easier than writing the equivalent procedural code, both in terms of ease-of-use and correctness. However, sometimes things don't behave the way you expect them to. Yep, a bug.

Most end-user bugs originate from handling the context wrong. Sometimes you forget what nesting level you are at, or you move things around without taking into account the nesting, thus breaking context-based expressions. The two utilities described below should help you out.

1.14.1 Probe

The Probe simply dumps information to the screen. It will help you inspect the context tree, the stream, and partially constructed objects, so you can understand your problem better. It has the same interface as any other field, and you can just stick it into a Struct, near the place you wish to inspect. Do note that the printout happens during the construction, before the final object is ready.

```
>>> d = Struct(
...     "count" / Byte,
...     "items" / Byte[this.count],
...     Probe(lookahead=32),
... )
>>> d.parse(b"\x05abcde\x01\x02\x03")

--------------------------------------------------
Probe, path is (parsing), into is None
Stream peek: (hexlified) b'010203'...
Container:
    count = 5
    items = ListContainer:
        97
        98
        99
        100
        101
--------------------------------------------------
```

There is also feature that looks inside the context and extracts a part of it using a lambda instead of printing the entire context.

```
>>> d = Struct(
...     "count" / Byte,
...     "items" / Byte[this.count],
...     Probe(this.count),
... )
>>> d.parse(b"\x05abcde\x01\x02\x03")

--------------------------------------------------
Probe, path is (parsing), into is this.count
5
--------------------------------------------------
```

1.14.2 Debugger

The Debugger is a pdb-based full python debugger. Unlike Probe, Debugger is a subconstruct (it wraps an inner construct), so you simply put it around the problematic construct. If no exception occurs, the return value is passed right through. Otherwise, an interactive debugger pops, letting you tweak around.

When an exception occurs while parsing, you can go up (using u) to the level of the debugger and set self.retval to the desired return value. This allows you to hot-fix the error. Then use q to quit the debugger prompt and resume normal execution with the fixed value. However, if you don't set self.retval, the exception will propagate up.

```
>>> Debugger(Byte[3]).build([])

--------------------------------------------------
Debugging exception of <Array: None>
path is (building)
  File "/media/arkadiusz/MAIN/GitHub/construct/construct/debug.py", line 192, in _
↪build
    return self.subcon._build(obj, stream, context, path)
  File "/media/arkadiusz/MAIN/GitHub/construct/construct/core.py", line 2149, in _
↪build
    raise RangeError("expected %d elements, found %d" % (count, len(obj)))
```

(continues on next page)

(continued from previous page)

```
construct.core.RangeError: expected 3 elements, found 0

> /media/arkadiusz/MAIN/GitHub/construct/construct/core.py(2149)_build()
-> raise RangeError("expected %d elements, found %d" % (count, len(obj)))
(Pdb) q
--------------------------------------------------
```

1.15 Compilation feature

Warning: This feature is fully implemented but may not be fully mature.

1.15.1 Overall

Construct 2.9 adds an experimental feature: compiling user made constructs into much faster (but less feature-rich) code. If you are familiar with Kaitai Struct, an alternative framework to Construct, Kaitai compiles yaml-based schemas into pure Python modules. Construct on the other hand, defines schemas in pure Python and compiles them into pure Python modules. Once you define a construct, you can use it to parse and build blobs without compilation. Compilation has only one purpose: performance.

It should be made clear that currently the compiler supports only parsing. Building and sizeof are deferred to original constructs, from which a compiled instance was made. Building support may be added in the future, depending on popularity of this feature. In that sense, perhaps the documentation should use the term "compiled parser" rather than "compiled construct".

Requirements

Compilation feature requires Construct 2.9, preferrably the newest version to date. More importantly, you should have a test suite of your own. Construct aims to be reliable, but the compiler makes some undocumented assumptions, and generates a code that "takes shortcuts". Since few checks are ommited by generated code, you should not use it to parse corrupted data.

Restrictions

Compiled classes only parse faster, building and sizeof defers to core classes

Sizeof is applied during compilation (not during parsing and building)

Exceptions do not include *path* information

Struct Sequence FocusedSeq Union LazyStruct do not support *_subcons _stream* context entries

Parsed hooks are not supported, ignored

1.15.2 Compiling schemas

Every construct (even those that do not compile) has a parameter-less *compile* method that returns also a construct (instance of Compiled class). It may be a good idea to compile something that is used for processing megabyte-sized data or millions of blobs. That compiled instance has *parse* and *build* methods just like the construct is was compiled from. Therefore, in your code, you can simply reassign the compiled instance over the original one.

```
>>> st = Struct("num" / Byte)
>>> st.parse(b"\x01")
Container(num=1)
>>> st = st.compile(filename="copyforinspection.py")
>>> st.parse(b"\x01")
Container(num=1)
```

Performance boost can be easily measured. This method also happens to be testing the correctness of the compiled parser, by making sure that both original and compiled instance parse into same results.

```
>>> print(st.benchmark(sampledata))
Timeit measurements:
parsing:           0.0000475557 sec/call
parsing compiled:  0.0000159182 sec/call
building:          0.0000591526 sec/call
```

1.15.3 Motivation

The code generated by compiler and core classes have essentially same functionality, but there is a noticable difference in performance. First half of performance boost is thanks to pre-processing, as shown in this chapter. Pre-processing means inserting constants instead of variable lookups, constants means just variables that are known at compile time. The second half is thanks to pypy. This chapter explains the performance difference by comparing *Struct FormatField BytesInteger Bytes* classes, including using the context. Example construct:

```
Struct(
    "num8" / Int8ub,
    "num24" / Int24ub,
    "data" / Bytes(this.num8),
)
```

Compiled parsing code:

```
def read_bytes(io, count):
    assert count >= 0
    data = io.read(count)
    assert len(data) == count
    return data
def parse_struct_1(io, this):
    this = Container(_ = this)
    try:
        this['num8'] = unpack('>B', read_bytes(io, 1))[0]
        this['num24'] = int.from_bytes(read_bytes(io, 3), byteorder='big',␣
↪signed=False)
        this['data'] = read_bytes(io, this.num8)
    except StopIteration:
        pass
    del this['_']
    return this
def parseall(io, this):
    return parse_struct_1(io, this)
compiledschema = Compiled(None, None, parseall)
```

Non-compiled parsing code:

```
def _read_stream(stream, length):
    if length < 0:
        raise StreamError("length must be non-negative, found %s" % length)
    try:
        data = stream.read(length)
    except Exception:
        raise StreamError("stream.read() failed, requested %s bytes" % (length,))
    if len(data) != length:
        raise StreamError("could not read enough bytes, expected %d, found %d" %␣
↪(length, len(data)))
    return data

class FormatField(Construct):
    def _parse(self, stream, context, path):
        data = _read_stream(stream, self.length)
        try:
            return struct.unpack(self.fmtstr, data)[0]
        except Exception:
            raise FormatFieldError("struct %r error during parsing" % self.fmtstr)

class BytesInteger(Construct):
    def _parse(self, stream, context, path):
        length = self.length(context) if callable(self.length) else self.length
        data = _read_stream(stream, length)
        if self.swapped:
            data = data[::-1]
        return bytes2integer(data, self.signed)

class Bytes(Construct):
    def _parse(self, stream, context, path):
        length = self.length(context) if callable(self.length) else self.length
        return _read_stream(stream, length)

class Renamed(Subconstruct):
    def _parse(self, stream, context, path):
        path += " -> %s" % (self.name,)
        return self.subcon._parse(stream, context, path)

class Struct(Construct):
    def _parse(self, stream, context, path):
        obj = Container()
        context = Container(_ = context)
        context._subcons = Container({sc.name:sc for sc in self.subcons if sc.name})
        for sc in self.subcons:
            try:
                subobj = sc._parse(stream, context, path)
                if sc.name:
                    obj[sc.name] = subobj
                    context[sc.name] = subobj
            except StopIteration:
                break
        return obj
```

There are several “shortcuts” that the compiled code does:

Function calls are relatively expensive, so an inlined expression is faster than a function returning the same exact expression. Therefore FormatField compiles into *struct.unpack(..., read_bytes(io, ...))* directly.

Literals like 1 and ‘>B’ are faster than object field lookup, dictionary lookup, or passing function arguments. Therefore

each instance of FormatField compiles into a similar expression but with different format-strings and byte-counts inlined, usually literals.

Passing parameters to functions is slower than just referring to variables in same scope. Therefore, for example, compiled Struct creates "this" variable that is accessible to all expressions generated by subcons, as it exists in same scope, but core Struct would call subcon._parse and pass entire context as parameter value, regardless whether that subcon even uses a context (for example FormatField VarInt have no need for a context). Its similar but not exactly the same with "restream" function. The lambda in second parameter is rebounding *io* to a different object (a stream that gets created inside restream function). On the other hand, *this* is not rebounded, it exists in outer scope.

If statement (or conditional ternary operator) with two possible expressions and a condition that could be evaluated at compile-time is slower than just one or the other expression. Therefore, for example, BytesInteger does a lookup to check if field is swapped, but compiled BytesInteger simply inlines 'big' or 'little' literal. Moreover, Struct checks if each subcon has a name and then inserts a value into the context dictionary, but compiled Struct simply has an assignment or not. This shortcut also applies to most constructs, those that accept context lambdas as parameters. Generated classes do not need to check if a parameter is a constant or a lambda, because what gets emitted is either something like "1" which is a literal, or something like "this.field" which is an object lookup. Both are valid expressions and evaluate without red tape, or checks.

Looping over an iterable is slower than a block of code that accesses each item once. The reason its slower is that each iteration must fetch another item, and also check termination condition. Loop unrolling technique requires the iterable (or list rather) to be known at compile-time, which is the case with Struct and Sequence instances. Therefore, compiled Struct emits one line per subcon, but core Struct loops over its subcons.

Function calls that only defer to another function are only wasting CPU cycles. This relates specifically to Renamed class, which in compiled code emits same code as its subcon. Entire functionality of Renamed class (maintaining path information) is not supported in compiled code, where it would serve as mere subconstruct, just deferring to subcon.

Building two identical dictionaries is slower than building just one. Struct maintains two dictionaries (called obj and context) which differ only by _ key, but compiled Struct maintains only one dictionary and removes the _ key before returning it.

This expressions (not lambdas) are expensive to compute in regular code but something like "this.field" in a compiled code is merely one object field lookup. Same applies to *len_ obj_ list_* expressions since they share the implementation with *this* expression.

Container is an implementation of so called AttrDict. It captures access to its attributes (field in this.field) and treats it as dictionary key access (this.field becomes this["field"]). However, due to internal CPython drawbacks, capturing attribute access involves some red tape, unlike accessing keys, which is done directly. Therefore compiled Struct emits lines that assign to Container keys, not attributes.

Empirical evidence

The "shortcuts" that are described above are not much, but amount to quite a large portion of actual run-time. In fact, they amount to about a third (31%) of entire run-time. Note that this benchmark includes only pure-python compile-time optimisations.

Notice that results are in microseconds (10**-6).

```
-------------------------------- benchmark: 158 tests --------------------------------
Name (time in us)                                  Min                 StdDev
--------------------------------------------------------------------------------------
test_class_array_parse                       284.7820 (74.05)       31.0403 (118.46)
test_class_array_parse_compiled               73.6430 (19.15)       10.7624 (41.07)
test_class_greedyrange_parse                 325.6610 (84.67)       31.8383 (121.50)
test_class_greedyrange_parse_compiled        300.9270 (78.24)       24.0149 (91.65)
test_class_repeatuntil_parse                  10.2730 (2.67)         0.8322 (3.18)
```

(continues on next page)

(continued from previous page)

```
test_class_repeatuntil_parse_compiled            7.3020 (1.90)          1.3155 (5.02)
test_class_string_parse                         21.2270 (5.52)          1.3555 (5.17)
test_class_string_parse_compiled                18.9030 (4.91)          1.6023 (6.11)
test_class_cstring_parse                        10.9060 (2.84)          1.0971 (4.19)
test_class_cstring_parse_compiled                9.4050 (2.45)          1.6083 (6.14)
test_class_pascalstring_parse                    7.9290 (2.06)          0.4959 (1.89)
test_class_pascalstring_parse_compiled           6.6670 (1.73)          0.6601 (2.52)
test_class_struct_parse                         43.5890 (11.33)         4.4993 (17.17)
test_class_struct_parse_compiled                18.7370 (4.87)          2.0198 (7.71)
test_class_sequence_parse                       20.7810 (5.40)          2.6298 (10.04)
test_class_sequence_parse_compiled              11.9820 (3.12)          3.2669 (12.47)
test_class_union_parse                          91.0570 (23.68)        10.2126 (38.97)
test_class_union_parse_compiled                 31.9240 (8.30)          3.5955 (13.72)
test_overall_parse                          3,200.7850 (832.23)       224.9197 (858.34)
test_overall_parse_compiled                 2,229.9610 (579.81)       118.2029 (451.09)
-------------------------------------------------------------------------------------------
```

1.15.4 Motivation, part 2

The second part of optimisation is just running the generated code on pypy. Since pypy is not using any type annotations, there is nothing to discuss in this chapter. The benchmark reflects the same code as in previous chapter, but ran on Pypy 2.7 rather than CPython 3.6.

Empirical evidence

Notice that results are in nanoseconds (10**-9).

```
---------------------------------- benchmark: 152 tests -----------------------------
↪---------
Name (time in ns)                                       Min                     StdDev
-------------------------------------------------------------------------------------
↪---------
test_class_array_parse                          11,042.9974 (103.52)       40,792.8559␣
↪(46.97)
test_class_array_parse_compiled                  9,088.0058 (85.20)        43,001.3909␣
↪(49.52)
test_class_greedyrange_parse                    14,402.0014 (135.01)       49,834.2047␣
↪(57.38)
test_class_greedyrange_parse_compiled            9,801.0059 (91.88)        39,296.4529␣
↪(45.25)
test_class_repeatuntil_parse                       318.4996 (2.99)          2,469.5524␣
↪(2.84)
test_class_repeatuntil_parse_compiled              309.3746 (2.90)        103,425.2134␣
↪(119.09)
test_class_string_parse                            966.8991 (9.06)        537,241.0095␣
↪(618.62)
test_class_string_parse_compiled                   726.6994 (6.81)          3,719.2657␣
↪(4.28)
test_class_cstring_parse                           782.2993 (7.33)          4,111.8970␣
↪(4.73)
test_class_cstring_parse_compiled                  591.1992 (5.54)        479,164.9746␣
↪(551.75)
test_class_pascalstring_parse                      465.0911 (4.36)          4,262.4397␣
↪(4.91)
```

(continues on next page)

(continued from previous page)

```
test_class_pascalstring_parse_compiled             298.4118 (2.80)         122,279.2150␣
↪(140.80)
test_class_struct_parse                          2,633.9985 (24.69)         14,654.3095␣
↪(16.87)
test_class_struct_parse_compiled                   949.7991 (8.90)           4,228.2890␣
↪(4.87)
test_class_sequence_parse                        1,310.6008 (12.29)          5,811.8046␣
↪(6.69)
test_class_sequence_parse_compiled                 732.2000 (6.86)           4,703.9483␣
↪(5.42)
test_class_union_parse                           5,619.9933 (52.69)         30,590.0630␣
↪(35.22)
test_class_union_parse_compiled                  2,699.9987 (25.31)         15,888.8206␣
↪(18.30)
test_overall_parse                           1,332,581.9891 (>1000.0)    2,274,995.4192␣
↪(>1000.0)
test_overall_parse_compiled                    690,380.0095 (>1000.0)      602,697.9721␣
↪(694.00)
-----------------------------------------------------------------------------------------
↪---------
```

1.15.5 Motivation, part 3

Warning: Benchmarks revealed that pypy makes the code run much faster than cython, therefore cython improvements were withdrawn, and compiler now generates pure python code that is compatible with Python 2 including pypy. This chapter is no longer relevant. It remained just for educational purposes.

This chapter talks about the second half of optimisation, which is due to Cython type annotations and type inference. I should state for the record, that I am no expert at Cython, and following explanatations are merely "the way I understand it". Please take that into account when reading it. Fourth example:

```
Struct(
    "num1" / Int8ul,
    "num2" / Int24ul,
    "fixedarray1" / Array(3, Int8ul),
    "name1" / CString("utf8"),
)
```

```
cdef bytes read_bytes(io, int count):
    if not count >= 0: raise StreamError
    cdef bytes data = io.read(count)
    if not len(data) == count: raise StreamError
    return data
cdef bytes parse_nullterminatedstring(io, int unitsize, bytes finalunit):
    cdef list result = []
    cdef bytes unit
    while True:
        unit = read_bytes(io, unitsize)
        if unit == finalunit:
            break
        result.append(unit)
    return b"".join(result)
```

(continues on next page)

(continued from previous page)

```
def parse_struct_1(io, this):
    this = Container(_ = this)
    try:
        this['num1'] = unpack('<B', read_bytes(io, 1))[0]
        this['num2'] = int.from_bytes(read_bytes(io, 3), byteorder='little',␣
↪signed=False)
        this['fixedarray1'] = ListContainer((unpack('<B', read_bytes(io, 1))[0]) for␣
↪i in range(3))
        this['name1'] = (parse_nullterminatedstring(io, 1, b'\x00')).decode('utf8')
        pass
    except StopIteration:
        pass
    del this['_']
    del this['_index']
    return this
def parseall(io, this):
    return parse_struct_1(io, this)
compiled = Compiled(None, None, parseall)
```

The primary cause of speedup in cython is this: if a variable is of known type, then operations on that variable can skip certain checks. If a variable is a pure python object, then those checks need to be added. A variable is considered of known type if either (1) its annotated like "cdef bytes data" or (2) its inferred like when using an annotated function call result like in "parse_nullterminatedstring(...).decode(...)" since "cdef bytes parse_nullterminatedstring(...)". If a variable is known to be a list, then calling "append" on it doesnt require checking if that object has such a method or matching signature (parameters). If a variable is known to be a bytes, then "len(data)" can be compiled into bytes-type length function, not a general-purpose length function that works on arbitrary objects, and also "unit == finalunit" can be compiled into bytes-type equality. If a variable is known to be a unicode, then ".decode('utf8')" can be compiled into str-type implementation. If cython knows that "struct.unpack" returns only tuples, then "...[0]" would compile into tuple-type getitem (index access). Examples are many, but the pattern is the same: type-specific code is faster than type-general code.

Second cause of speedup is due to special handling of integers. While most annotations like "cdef bytes" refer to specific albeit Python types, the "cdef int" actually does not refer to any Python type. It represents a C-integer which is allocated on the stack or in registers, unlike the other types which are allocated on the heap. All operations on C-integers are therefore much faster than on Python-integers. In example code, this affects "count >= 0" and "len(data) == count".

Empirical evidence

Below micro-benchmarks show the difference between core classes and cython-compiled classes. Only those where performance boost was highest are listed (although they also happen to be the most important), some other classes have little speedup, and some have none.

Notice that results are in microseconds (10**-6).

```
------------------------------ benchmark: 152 tests ------------------------------
Name (time in us)                                  Min                StdDev
----------------------------------------------------------------------------------
test_class_array_parse                        286.5460 (73.85)     42.8831 (89.84)
test_class_array_parse_compiled                30.7200 (7.92)       6.9577 (14.58)
test_class_greedyrange_parse                  320.9860 (82.73)     45.9480 (96.26)
test_class_greedyrange_parse_compiled         262.7010 (67.71)     36.4504 (76.36)
test_class_repeatuntil_parse                   10.1850 (2.63)       2.4147 (5.06)
test_class_repeatuntil_parse_compiled           6.8880 (1.78)       1.5471 (3.24)
test_class_string_parse                        20.4400 (5.27)       4.4044 (9.23)
```

(continues on next page)

(continued from previous page)

```
test_class_string_parse_compiled                 9.1470 (2.36)       2.2427 (4.70)
test_class_cstring_parse                        11.2290 (2.89)       1.6216 (3.40)
test_class_cstring_parse_compiled                5.6080 (1.45)       1.0321 (2.16)
test_class_pascalstring_parse                    7.8560 (2.02)       1.8567 (3.89)
test_class_pascalstring_parse_compiled           5.8910 (1.52)       0.9466 (1.98)
test_class_struct_parse                         44.1300 (11.37)      6.8434 (14.34)
test_class_struct_parse_compiled                16.9070 (4.36)       3.0500 (6.39)
test_class_sequence_parse                       21.5420 (5.55)       2.6852 (5.63)
test_class_sequence_parse_compiled              10.1530 (2.62)       2.1645 (4.53)
test_class_union_parse                          91.9150 (23.69)     10.7812 (22.59)
test_class_union_parse_compiled                 22.5970 (5.82)      15.2649 (31.98)
test_overall_parse                          2,126.2570 (548.01)   255.0154 (534.27)
test_overall_parse_compiled                 1,124.9560 (289.94)   127.4730 (267.06)
-------------------------------------------------------------------------------------
```

1.15.6 Comparison with Kaitai Struct

Kaitai Struct is a very respectable competitor, so I believe a benchmark-based comparison should be presented. Construct and Kaitai have very different capabilities: Kaitai supports about a dozen languages, Construct only supports Python, Kaitai offers only basic common features, Construct offers python-only stuff like Numpy and Pickle support, Kaitai does only parsing, Construct does also building. In a sense, those libraries are in two different categories (like sumo and karate). There are multiple scenarios where either library would not be usable.

Example used for comparison:

```
Struct(
    "count" / Int32ul,
    "items" / Array(this.count, Struct(
        "num1" / Int8ul,
        "num2" / Int24ul,
        "flags" / BitStruct(
            "bool1" / Flag,
            "num4" / BitsInteger(3),
            Padding(4),
        ),
        "fixedarray1" / Array(3, Int8ul),
        "name1" / CString("utf8"),
        "name2" / PascalString(Int8ul, "utf8"),
    )),
)
```

```
meta:
  id: comparison_1_kaitai
  encoding: utf-8
  endian: le
seq:
  - id: count
    type: u4
  - id: items
    repeat: expr
    repeat-expr: count
    type: item
types:
  item:
    seq:
```

(continues on next page)

(continued from previous page)

```
      - id: num1
        type: u1
      - id: num2_lo
        type: u2
      - id: num2_hi
        type: u1
      - id: flags
        type: flags
      - id: fixedarray1
        repeat: expr
        repeat-expr: 3
        type: u1
      - id: name1
        type: strz
      - id: len_name2
        type: u1
      - id: name2
        type: str
        size: len_name2
    instances:
      num2:
        value: 'num2_hi << 16 | num2_lo'
    types:
      flags:
        seq:
          - id: bool1
            type: b1
          - id: num4
            type: b3
          - id: padding
            type: b4
```

Suprisingly, Kaitai won the benchmark! Honestly, I am shocked and dismayed that it did. The only explanation that I can point out, is that Kaitai is parsing structs into class objects (with attributes) while Construct parses into dictionaries (with keys). However that one detail seems unlikely explanation for the huge discrepancy in benchmark results. Perhaps there is a flaw in the methodology. But until that is proven, Kaitai gets its respects. Congrats.

```
$ python3.6 comparison_1_construct.py
Timeit measurements:
parsing:           0.1024609069 sec/call
parsing compiled:  0.0410809368 sec/call

$ pypy comparison_1_construct.py
Timeit measurements:
parsing:           0.0108308416 sec/call
parsing compiled:  0.0062594243 sec/call
```

```
$ python3.6 comparison_1_kaitai.py
Timeit measurements:
parsing:           0.0250326035 sec/call

$ pypy comparison_1_kaitai.py
Timeit measurements:
parsing:           0.0019435351 sec/call
```

CHAPTER 2

API Reference

2.1 Core API: Abstract classes

class construct.**Construct**

The mother of all constructs.

This object is generally not directly instantiated, and it does not directly implement parsing and building, so it is largely only of interest to subclass implementors. There are also other abstract classes sitting on top of this one.

The external user API:

- *parse*
- *parse_stream*
- *parse_file*
- *build*
- *build_stream*
- *build_file*
- *sizeof*
- *compile*
- *benchmark*

Subclass authors should not override the external methods. Instead, another API is available:

- *_parse*
- *_build*
- *_sizeof*
- *_actualsize*
- *_emitparse*

- *_emitbuild*
- *__getstate__*
- *__setstate__*

Attributes and Inheritance:

All constructs have a name and flags. The name is used for naming struct members and context dictionaries. Note that the name can be a string, or None by default. A single underscore "_" is a reserved name, used as up-level in nested containers. The name should be descriptive, short, and valid as a Python identifier, although these rules are not enforced. The flags specify additional behavioral information about this construct. Flags are used by enclosing constructs to determine a proper course of action. Flags are often inherited from inner subconstructs but that depends on each class.

benchmark(*sampledata*, *filename=None*)

Measures performance of your construct (its parsing and building runtime), both for the original instance and the compiled instance. Uses timeit module, over at min 1 sample, and at max over 1 second time.

Optionally, results are saved to a text file for later inspection. Otherwise you can print the result string to terminal.

Also this method checks correctness, by comparing parsing/building results from both instances.

Parameters

- **sampledata** – bytes, a valid blob parsable by this construct
- **filename** – optional, string, source is saved to that file

Returns string containing measurements

build(*obj*, ***contextkw*)

Build an object in memory (a bytes object).

Whenever data cannot be written, ConstructError or its derivative is raised. This method is NOT ALLOWED to raise any other exceptions although (1) user-defined lambdas can raise arbitrary exceptions which are propagated (2) external libraries like numpy can raise arbitrary exceptions which are propagated (3) some list and dict lookups can raise IndexError and KeyError which are propagated.

Context entries are passed only as keyword parameters **contextkw.

Parameters ****contextkw** – context entries, usually empty

Returns bytes

Raises ***ConstructError*** – raised for any reason

build_file(*obj*, *filename*, ***contextkw*)

Build an object into a closed binary file. See build().

build_stream(*obj*, *stream*, ***contextkw*)

Build an object directly into a stream. See build().

compile(*filename=None*)

Transforms a construct into another construct that does same thing (has same parsing and building semantics) but is much faster when parsing. Already compiled instances just compile into itself.

Optionally, partial source code can be saved to a text file. This is meant only to inspect the generated code, not to import it from external scripts.

Returns Compiled instance

parse(*data*, ***contextkw*)
Parse an in-memory buffer (often bytes object). Strings, buffers, memoryviews, and other complete buffers can be parsed with this method.

Whenever data cannot be read, ConstructError or its derivative is raised. This method is NOT ALLOWED to raise any other exceptions although (1) user-defined lambdas can raise arbitrary exceptions which are propagated (2) external libraries like numpy can raise arbitrary exceptions which are propagated (3) some list and dict lookups can raise IndexError and KeyError which are propagated.

Context entries are passed only as keyword parameters **contextkw.

Parameters ****contextkw** – context entries, usually empty

Returns some value, usually based on bytes read from the stream but sometimes it is computed from nothing or from the context dictionary, sometimes its non-deterministic

Raises ***ConstructError*** – raised for any reason

parse_file(*filename*, ***contextkw*)
Parse a closed binary file. See parse().

parse_stream(*stream*, ***contextkw*)
Parse a stream. Files, pipes, sockets, and other streaming sources of data are handled by this method. See parse().

sizeof(***contextkw*)
Calculate the size of this object, optionally using a context.

Some constructs have fixed size (like FormatField), some have variable-size and can determine their size given a context entry (like Bytes(this.otherfield1)), and some cannot determine their size (like VarInt).

Whenever size cannot be determined, SizeofError is raised. This method is NOT ALLOWED to raise any other exception, even if eg. context dictionary is missing a key, or subcon propagates ConstructError-derivative exception.

Context entries are passed only as keyword parameters **contextkw.

Parameters ****contextkw** – context entries, usually empty

Returns integer if computable, SizeofError otherwise

Raises ***SizeofError*** – size could not be determined in actual context, or is impossible to be determined

class construct.**Subconstruct**(*subcon*)
Abstract subconstruct (wraps an inner construct, inheriting its name and flags). Parsing and building is by default deferred to subcon, same as sizeof.

Parameters **subcon** – Construct instance

class construct.**Adapter**(*subcon*)
Abstract adapter class.

Needs to implement *_decode()* for parsing and *_encode()* for building.

Parameters **subcon** – Construct instance

class construct.**SymmetricAdapter**(*subcon*)
Abstract adapter class.

Needs to implement *_decode()* only, for both parsing and building.

Parameters **subcon** – Construct instance

class construct.**Validator**(*subcon*)
Abstract class that validates a condition on the encoded/decoded object.

Needs to implement *_validate()* that returns a bool (or a truthy value)

Parameters **subcon** – Construct instance

class construct.**Tunnel**(*subcon*)
Abstract class that allows other constructs to read part of the stream as if they were reading the entrie stream. See Prefixed for example.

Needs to implement *_decode()* for parsing and *_encode()* for building.

class construct.**Compiled**(*source*, *defersubcon*, *parsefunc*)
Used internally.

benchmark(*sampledata*, *filename=None*)
Measures performance of your construct (its parsing and building runtime), both for the original instance and the compiled instance. Uses timeit module, over at min 1 sample, and at max over 1 second time.

Optionally, results are saved to a text file for later inspection. Otherwise you can print the result string to terminal.

Also this method checks correctness, by comparing parsing/building results from both instances.

Parameters

- **sampledata** – bytes, a valid blob parsable by this construct
- **filename** – optional, string, source is saved to that file

Returns string containing measurements

compile(*filename=None*)
Transforms a construct into another construct that does same thing (has same parsing and building semantics) but is much faster when parsing. Already compiled instances just compile into itself.

Optionally, partial source code can be saved to a text file. This is meant only to inspect the generated code, not to import it from external scripts.

Returns Compiled instance

2.2 Core API: Exception types

construct.**ConstructError**()
Common base class for all non-exit exceptions.

construct.**SizeofError**()
Common base class for all non-exit exceptions.

construct.**AdaptationError**()
Common base class for all non-exit exceptions.

construct.**ValidationError**()
Common base class for all non-exit exceptions.

construct.**StreamError**()
Common base class for all non-exit exceptions.

construct.**FormatFieldError**()
Common base class for all non-exit exceptions.

construct.**IntegerError**()
Common base class for all non-exit exceptions.

construct.**StringError**()
Common base class for all non-exit exceptions.

construct.**MappingError**()
Common base class for all non-exit exceptions.

construct.**RangeError**()
Common base class for all non-exit exceptions.

construct.**RepeatError**()
Common base class for all non-exit exceptions.

construct.**ConstError**()
Common base class for all non-exit exceptions.

construct.**IndexFieldError**()
Common base class for all non-exit exceptions.

construct.**CheckError**()
Common base class for all non-exit exceptions.

construct.**ExplicitError**()
Common base class for all non-exit exceptions.

construct.**NamedTupleError**()
Common base class for all non-exit exceptions.

construct.**TimestampError**()
Common base class for all non-exit exceptions.

construct.**UnionError**()
Common base class for all non-exit exceptions.

construct.**SelectError**()
Common base class for all non-exit exceptions.

construct.**SwitchError**()
Common base class for all non-exit exceptions.

construct.**StopFieldError**()
Common base class for all non-exit exceptions.

construct.**PaddingError**()
Common base class for all non-exit exceptions.

construct.**TerminatedError**()
Common base class for all non-exit exceptions.

construct.**RawCopyError**()
Common base class for all non-exit exceptions.

construct.**ChecksumError**()
Common base class for all non-exit exceptions.

construct.**CancelParsing**()
Common base class for all non-exit exceptions.

2.3 Core API: Bytes and bits

construct.**Bytes**(*length*)

Field consisting of a specified number of bytes.

Parses into a bytes (of given length). Builds into the stream directly (but checks that given object matches specified length). Can also build from an integer for convenience (although BytesInteger should be used instead). Size is the specified length.

Parameters **length** – integer or context lambda

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- ***StringError*** – building from non-bytes value, perhaps unicode

Can propagate any exception from the lambda, possibly non-ConstructError.

Example:

```
>>> d = Bytes(4)
>>> d.parse(b'beef')
b'beef'
>>> d.build(b'beef')
b'beef'
>>> d.build(0)
b'\x00\x00\x00\x00'
>>> d.sizeof()
4

>>> d = Struct(
...     "length" / Int8ub,
...     "data" / Bytes(this.length),
... )
>>> d.parse(b"\x04beef")
Container(length=4)(data=b'beef')
>>> d.sizeof()
construct.core.SizeofError: cannot calculate size, key not found in context
```

construct.**GreedyBytes**()

Field consisting of unknown number of bytes.

Parses the stream to the end. Builds into the stream directly (without checks). Size is undefined.

Raises

- ***StreamError*** – stream failed when reading until EOF
- ***StringError*** – building from non-bytes value, perhaps unicode

Example:

```
>>> GreedyBytes.parse(b"asislight")
b'asislight'
>>> GreedyBytes.build(b"asislight")
b'asislight'
```

construct.**setGlobalPrintFullStrings**(*enabled=False*)

When enabled, Container __str__ produces full content of bytes and unicode strings, otherwise and by default, it produces truncated output (16 bytes and 32 characters).

Parameters **enabled** – bool

construct.**Bitwise**(*subcon*)

Converts the stream from bytes to bits, and passes the bitstream to underlying subcon. Bitstream is a stream that contains 8 times as many bytes, and each byte is either \x00 or \x01 (in documentation those bytes are called bits).

Parsing building and size are deferred to subcon, although size gets divided by 8.

Parameters **subcon** – Construct instance, any field that works with bits (like BitsInteger) or is bit-byte agnostic (like Struct or Flag)

See *Transformed* and *Restreamed* for raisable exceptions.

Example:

```
>>> d = Bitwise(Struct(
...     'a' / Nibble,
...     'b' / Bytewise(Float32b),
...     'c' / Padding(4),
... ))
>>> d.parse(bytes(5))
Container(a=0)(b=0.0)(c=None)
>>> d.sizeof()
5
```

construct.**Bytewise**(*subcon*)

Converts the bitstream back to normal byte stream. Must be used within *Bitwise*.

Parsing building and size are deferred to subcon, although size gets multiplied by 8.

Parameters **subcon** – Construct instance, any field that works with bytes or is bit-byte agnostic

See *Transformed* and *Restreamed* for raisable exceptions.

Example:

```
>>> d = Bitwise(Struct(
...     'a' / Nibble,
...     'b' / Bytewise(Float32b),
...     'c' / Padding(4),
... ))
>>> d.parse(bytes(5))
Container(a=0)(b=0.0)(c=None)
>>> d.sizeof()
5
```

2.4 Core API: Integers and Floats

construct.**FormatField**(*endianity*, *format*)

Field that uses *struct* module to pack and unpack CPU-sized integers and floats. This is used to implement most Int* Float* fields, but for example cannot pack 24-bit integers, which is left to *BytesInteger* class.

See struct module documentation for instructions on crafting format strings.

Parses into an integer. Builds from an integer into specified byte count and endianness. Size is determined by *struct* module according to specified format string.

Parameters

- **endianity** – string, character like: < > =
- **format** – string, character like: f d B H L Q b h l q

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- ***FormatFieldError*** – wrong format string, or struct.(un)pack complained about the value

Example:

```
>>> d = FormatField(">", "H") or Int16ub
>>> d.parse(b"\x01\x00")
256
>>> d.build(256)
b"\x01\x00"
>>> d.sizeof()
2
```

construct.**BytesInteger**(*length*, *signed=False*, *swapped=False*)

Field that packs arbitrarily large integers. Some Int24* fields use this class.

Parses into an integer. Builds from an integer into specified byte count and endianness. Size is specified in ctor.

Analog to *BitsInteger* that operates on bits. In fact, `BytesInteger(n)` is equivalent to `Bitwise(BitsInteger(8*n))` and `BitsInteger(n)` is equivalent to `Bytewise(BytesInteger(n//8))`.

Parameters

- **length** – integer or context lambda, number of bytes in the field
- **signed** – bool, whether the value is signed (two's complement), default is False (unsigned)
- **swapped** – bool, whether to swap byte order (little endian), default is False (big endian)

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- ***IntegerError*** – lenght is negative, given a negative value when field is not signed, or not an integer

Can propagate any exception from the lambda, possibly non-ConstructError.

Example:

```
>>> d = BytesInteger(4) or Int32ub
>>> d.parse(b"abcd")
1633837924
>>> d.build(1)
b'\x00\x00\x00\x01'
>>> d.sizeof()
4
```

construct.**BitsInteger**(*length*, *signed=False*, *swapped=False*)

Field that packs arbitrarily large (or small) integers. Some fields (Bit Nibble Octet) use this class. Must be enclosed in *Bitwise* context.

Parses into an integer. Builds from an integer into specified bit count and endianness. Size (in bits) is specified in ctor.

Note that little-endianness is only defined for multiples of 8 bits.

Analog to *BytesInteger* that operates on bytes. In fact, `BytesInteger(n)` is equivalent to `Bitwise(BitsInteger(8*n))` and `BitsInteger(n)` is equivalent to `Bytewise(BytesInteger(n//8))`.

Parameters

- **length** – integer or context lambda, number of bits in the field
- **signed** – bool, whether the value is signed (two's complement), default is False (unsigned)
- **swapped** – bool, whether to swap byte order (little endian), default is False (big endian)

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- ***IntegerError*** – lenght is negative, given a negative value when field is not signed, or not an integer

Can propagate any exception from the lambda, possibly non-ConstructError.

Example:

```
>>> d = Bitwise(BitsInteger(8)) or Bitwise(Octet)
>>> d.parse(b"\x10")
16
>>> d.build(255)
b'\xff'
>>> d.sizeof()
1
```

`construct.`**`VarInt`**`()`

VarInt encoded integer. Each 7 bits of the number are encoded in one byte of the stream, where leftmost bit (MSB) is unset when byte is terminal. Scheme is defined at Google site related to Protocol Buffers.

Can only encode non-negative numbers.

Parses into an integer. Builds from an integer. Size is undefined.

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- ***IntegerError*** – given a negative value, or not an integer

Example:

```
>>> VarInt.build(1)
b'\x01'
>>> VarInt.build(2**100)
b'\x80\x80\x80\x80\x80\x80\x80\x80\x80\x80\x80\x80\x80\x80\x80\x04'
```

2.5 Core API: Strings

construct.core.**possiblestringencodings = {'ascii': 1, 'u16': 2, 'u32': 4, 'u8': 1, 'uti**
dict() -> new empty dictionary dict(mapping) -> new dictionary initialized from a mapping object's

(key, value) pairs

dict(iterable) -> new dictionary initialized as if via: d = {} for k, v in iterable:

d[k] = v

dict(kwargs) -> new dictionary initialized with the name=value pairs** in the keyword argument list. For example: dict(one=1, two=2)

construct.**PaddedString**(*length*, *encoding*)
Configurable, fixed-length or variable-length string field.

When parsing, the byte string is stripped of null bytes (per encoding unit), then decoded. Length is an integer or context lambda. When building, the string is encoded and then padded to specified length. If encoded string is larger than the specified length, it fails with PaddingError. Size is same as length parameter.

> **Warning:** PaddedString and CString only support encodings explicitly listed in *possiblestringencodings*.

Parameters

- **length** – integer or context lambda, length in bytes (not unicode characters)
- **encoding** – string like: utf8 utf16 utf32 ascii

Raises

- ***StringError*** – building a non-unicode string
- ***StringError*** – selected encoding is not on supported list

Can propagate any exception from the lambda, possibly non-ConstructError.

Example:

```
>>> d = PaddedString(10, "utf8")
>>> d.build(u"")
b'\xd0\x90\xd1\x84\xd0\xbe\xd0\xbd\x00\x00'
>>> d.parse(_)
u''
```

construct.**PascalString**(*lengthfield*, *encoding*)
Length-prefixed string. The length field can be variable length (such as VarInt) or fixed length (such as Int64ub). `VarInt` is recommended when designing new protocols. Stored length is in bytes, not characters. Size is not defined.

Parameters

- **lengthfield** – Construct instance, field used to parse and build the length (like VarInt Int64ub)
- **encoding** – string like: utf8 utf16 utf32 ascii

Raises ***StringError*** – building a non-unicode string

Example:

```
>>> d = PascalString(VarInt, "utf8")
>>> d.build(u"")
b'\x08\xd0\x90\xd1\x84\xd0\xbe\xd0\xbd'
>>> d.parse(_)
u''
```

construct.**CString**(*encoding*)

String ending in a terminating null byte (or null bytes in case of UTF16 UTF32).

Warning: String and CString only support encodings explicitly listed in *possiblestringencodings* .

Parameters **encoding** – string like: utf8 utf16 utf32 ascii

Raises

- ***StringError*** – building a non-unicode string
- ***StringError*** – selected encoding is not on supported list

Example:

```
>>> d = CString("utf8")
>>> d.build(u"")
b'\xd0\x90\xd1\x84\xd0\xbe\xd0\xbd\x00'
>>> d.parse(_)
u''
```

construct.**GreedyString**(*encoding*)

String that reads entire stream until EOF, and writes a given string as-is. Analog to `GreedyBytes` but also applies unicode-to-bytes encoding.

Parameters **encoding** – string like: utf8 utf16 utf32 ascii

Raises

- ***StringError*** – building a non-unicode string
- ***StreamError*** – stream failed when reading until EOF

Example:

```
>>> d = GreedyString("utf8")
>>> d.build(u"")
b'\xd0\x90\xd1\x84\xd0\xbe\xd0\xbd'
>>> d.parse(_)
u''
```

construct.**setGlobalPrintFullStrings**(*enabled=False*)

When enabled, Container __str__ produces full content of bytes and unicode strings, otherwise and by default, it produces truncated output (16 bytes and 32 characters).

Parameters **enabled** – bool

2.6 Core API: Mappings

construct.**Flag**()

One byte (or one bit) field that maps to True or False. Other non-zero bytes are also considered True. Size is defined as 1.

Raises ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes

Example:

```
>>> Flag.parse(b"\x01")
True
>>> Flag.build(True)
b'\x01'
```

construct.**Enum**(*subcon*, **merge*, ***mapping*)

Translates unicode label names to subcon values, and vice versa.

Parses integer subcon, then uses that value to lookup mapping dictionary. Returns an integer-convertible string (if mapping found) or an integer (otherwise). Building is a reversed process. Can build from an integer flag or string label. Size is same as subcon, unless it raises SizeofError.

There is no default parameter, because if no mapping is found, it parses into an integer without error.

This class supports enum34 module. See examples.

This class supports exposing member labels as attributes, as integer-convertible strings. See examples.

Parameters

- **subcon** – Construct instance, subcon to map to/from
- ***merge** – optional, list of enum.IntEnum and enum.IntFlag instances, to merge labels and values from
- ****mapping** – dict, mapping string names to values

Raises ***MappingError*** – building from string but no mapping found

Example:

```
>>> d = Enum(Byte, one=1, two=2, four=4, eight=8)
>>> d.parse(b"\x01")
'one'
>>> int(d.parse(b"\x01"))
1
>>> d.parse(b"\xff")
255
>>> int(d.parse(b"\xff"))
255

>>> d.build(d.one or "one" or 1)
b'\x01'
>>> d.one
'one'

import enum
class E(enum.IntEnum or enum.IntFlag):
    one = 1
    two = 2
```

(continues on next page)

(continued from previous page)

```
Enum(Byte, E) <--> Enum(Byte, one=1, two=2)
FlagsEnum(Byte, E) <--> FlagsEnum(Byte, one=1, two=2)
```

construct.**FlagsEnum**(*subcon*, **merge*, ***flags*)

Translates unicode label names to subcon integer (sub)values, and vice versa.

Parses integer subcon, then creates a Container, where flags define each key. Builds from a container by bitwise-oring of each flag if it matches a set key. Can build from an integer flag or string label directly, as well as | concatenations thereof (see examples). Size is same as subcon, unless it raises SizeofError.

This class supports enum34 module. See examples.

This class supports exposing member labels as attributes, as bitwisable strings. See examples.

Parameters

- **subcon** – Construct instance, must operate on integers
- ***merge** – optional, list of enum.IntEnum and enum.IntFlag instances, to merge labels and values from
- ****flags** – dict, mapping string names to integer values

Raises

- ***MappingError*** – building from object not like: integer string dict
- ***MappingError*** – building from string but no mapping found

Can raise arbitrary exceptions when computing | and & and value is non-integer.

Example:

```
>>> d = FlagsEnum(Byte, one=1, two=2, four=4, eight=8)
>>> d.parse(b"\x03")
Container(one=True, two=True, four=False, eight=False)
>>> d.build(dict(one=True,two=True))
b'\x03'

>>> d.build(d.one|d.two or "one|two" or 1|2)
b'\x03'

import enum
class E(enum.IntEnum or enum.IntFlag):
    one = 1
    two = 2

Enum(Byte, E) <--> Enum(Byte, one=1, two=2)
FlagsEnum(Byte, E) <--> FlagsEnum(Byte, one=1, two=2)
```

construct.**setGlobalPrintFalseFlags**(*enabled=False*)

When enabled, Container __str__ that was produced by FlagsEnum parsing prints all values, otherwise and by default, it prints only the values that are True.

Parameters **enabled** – bool

construct.**Mapping**(*subcon*, *mapping*)

Adapter that maps objects to other objects. Translates objects after parsing and before building. Can for example, be used to translate between enum34 objects and strings, but Enum class supports enum34 already and is recommended.

Parameters

- **subcon** – Construct instance
- **mapping** – dict, for encoding (building) mapping, the reversed is used for parsing mapping

Raises ***MappingError*** – parsing or building but no mapping found

Example:

```
>>> x = object
>>> d = Mapping(Byte, {x:0})
>>> d.parse(b"\x00")
x
>>> d.build(x)
b'\x00'
```

2.7 Core API: Structs and Sequences

construct.**Struct**(**subcons*, ***subconskw*)

Sequence of usually named constructs, similar to structs in C. The members are parsed and build in the order they are defined. If a member is anonymous (its name is None) then it gets parsed and the value discarded, or it gets build from nothing (from None).

Some fields do not need to be named, since they are built without value anyway. See: Const Padding Check Error Pass Terminated Seek Tell for examples of such fields. *Embedded* fields do not need to (and should not) be named.

Operator + can also be used to make Structs (although not recommended).

Parses into a Container (dict with attribute and key access) where keys match subcon names. Builds from a dict (not necessarily a Container) where each member gets a value from the dict matching the subcon name. If field has build-from-none flag, it gets build even when there is no matching entry in the dict. Size is the sum of all subcon sizes, unless any subcon raises SizeofError.

This class does context nesting, meaning its members are given access to a new dictionary where the "_" entry points to the outer context. When parsing, each member gets parsed and subcon parse return value is inserted into context under matching key only if the member was named. When building, the matching entry gets inserted into context before subcon gets build, and if subcon build returns a new value (not None) that gets replaced in the context.

This class supports embedding. *Embedded* semantics dictate, that during instance creation (in ctor), each field is checked for embedded flag, and its subcon members are merged. This changes behavior of some code examples. Only few classes are supported: Struct Sequence FocusedSeq Union LazyStruct, although those can be used interchangably (a Struct can embed a Sequence, or rather its members).

This class exposes subcons as attributes. You can refer to subcons that were inlined (and therefore do not exist as variable in the namespace) by accessing the struct attributes, under same name. Also note that compiler does not support this feature. See examples.

This class exposes subcons in the context. You can refer to subcons that were inlined (and therefore do not exist as variable in the namespace) within other inlined fields using the context. Note that you need to use a lambda (*this* expression is not supported). Also note that compiler does not support this feature. See examples.

This class supports stopping. If *StopIf* field is a member, and it evaluates its lambda as positive, this class ends parsing or building as successful without processing further fields.

Parameters

- ***subcons** – Construct instances, list of members, some can be anonymous
- ****subconskw** – Construct instances, list of members (requires Python 3.6)

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- **KeyError** – building a subcon but found no corresponding key in dictionary

Example:

```
>>> d = Struct("num"/Int8ub, "data"/Bytes(this.num))
>>> d.parse(b"\x04DATA")
Container(num=4)(data=b"DATA")
>>> d.build(dict(num=4, data=b"DATA"))
b"\x04DATA"

>>> d = Struct(Const(b"MZ"), Padding(2), Pass, Terminated)
>>> d.build({})
b'MZ\x00\x00'
>>> d.parse(_)
Container()
>>> d.sizeof()
4

>>> d = Struct(
...     "animal" / Enum(Byte, giraffe=1),
... )
>>> d.animal.giraffe
'giraffe'
>>> d = Struct(
...     "count" / Byte,
...     "data" / Bytes(lambda this: this.count - this._subcons.count.sizeof()),
... )
>>> d.build(dict(count=3, data=b"12"))
b'\x0312'

Alternative syntax (not recommended):
>>> ("a"/Byte + "b"/Byte + "c"/Byte + "d"/Byte)

Alternative syntax, but requires Python 3.6 or any PyPy:
>>> Struct(a=Byte, b=Byte, c=Byte, d=Byte)
```

construct.**Sequence**(**subcons*, ***subconskw*)

Sequence of usually un-named constructs. The members are parsed and build in the order they are defined. If a member is named, its parsed value gets inserted into the context. This allows using members that refer to previous members. *Embedded* fields do not need to (and should not) be named.

Operator >> can also be used to make Sequences (although not recommended).

Parses into a ListContainer (list with pretty-printing) where values are in same order as subcons. Builds from a list (not necessarily a ListContainer) where each subcon is given the element at respective position. Size is the sum of all subcon sizes, unless any subcon raises SizeofError.

This class does context nesting, meaning its members are given access to a new dictionary where the "_" entry points to the outer context. When parsing, each member gets parsed and subcon parse return value is inserted into context under matching key only if the member was named. When building, the matching entry gets inserted into context before subcon gets build, and if subcon build returns a new value (not None) that gets replaced in the context.

This class supports embedding. *Embedded* semantics dictate, that during instance creation (in ctor), each field is checked for embedded flag, and its subcon members are merged. This changes behavior of some code examples. Only few classes are supported: Struct Sequence FocusedSeq Union LazyStruct, although those can be used interchangably (a Struct can embed a Sequence, or rather its members).

This class exposes subcons as attributes. You can refer to subcons that were inlined (and therefore do not exist as variable in the namespace) by accessing the struct attributes, under same name. Also note that compiler does not support this feature. See examples.

This class exposes subcons in the context. You can refer to subcons that were inlined (and therefore do not exist as variable in the namespace) within other inlined fields using the context. Note that you need to use a lambda (*this* expression is not supported). Also note that compiler does not support this feature. See examples.

This class supports stopping. If *StopIf* field is a member, and it evaluates its lambda as positive, this class ends parsing or building as successful without processing further fields.

Parameters

- ***subcons** – Construct instances, list of members, some can be named
- ****subconskw** – Construct instances, list of members (requires Python 3.6)

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- **KeyError** – building a subcon but found no corresponding key in dictionary

Example:

```
>>> d = Sequence(Byte, Float32b)
>>> d.build([0, 1.23])
b'\x00?\x9dp\xa4'
>>> d.parse(_)
[0, 1.2300000190734863] # a ListContainer

>>> d = Sequence(
...     "animal" / Enum(Byte, giraffe=1),
... )
>>> d.animal.giraffe
'giraffe'
>>> d = Sequence(
...     "count" / Byte,
...     "data" / Bytes(lambda this: this.count - this._subcons.count.sizeof()),
... )
>>> d.build([3, b"12"])
b'\x0312'

Alternative syntax (not recommended):
>>> (Byte >> "Byte >> "c"/Byte >> "d"/Byte)

Alternative syntax, but requires Python 3.6 or any PyPy:
>>> Sequence(a=Byte, b=Byte, c=Byte, d=Byte)
```

construct.**Embedded**(*subcon*)

Special wrapper that allows outer multiple-subcons construct to merge fields from another multiple-subcons construct. Embedded does not change a field, only wraps it like a candy with a flag.

Warning: Can only be used between Struct Sequence FocusedSeq Union LazyStruct, although they can be used interchangably, for example Struct can embed fields from a Sequence. There is also *EmbeddedSwitch* macro that pseudo-embeds a Switch. Its not possible to embed IfThenElse.

Parsing building and sizeof are deferred to subcon.

Parameters **subcon** – Construct instance, its fields to embed inside a struct or sequence

Example:

```
>>> outer = Struct(
...     Embedded(Struct(
...         "data" / Bytes(4),
...     )),
... )
>>> outer.parse(b"1234")
Container(data=b'1234')
```

construct.**AlignedStruct**(*modulus*, **subcons*, ***subconskw*)

Makes a structure where each field is aligned to the same modulus (it is a struct of aligned fields, NOT an aligned struct).

See *Aligned* and *Struct* for semantics and raisable exceptions.

Parameters

- **modulus** – integer or context lambda, passed to each member
- ***subcons** – Construct instances, list of members, some can be anonymous
- ****subconskw** – Construct instances, list of members (requires Python 3.6)

Example:

```
>>> d = AlignedStruct(4, "a"/Int8ub, "b"/Int16ub)
>>> d.build(dict(a=0xFF,b=0xFFFF))
b'\xff\x00\x00\x00\xff\xff\x00\x00'
```

construct.**BitStruct**(**subcons*, ***subconskw*)

Makes a structure inside a Bitwise.

See *Bitwise* and *Struct* for semantics and raisable exceptions.

Parameters

- ***subcons** – Construct instances, list of members, some can be anonymous
- ****subconskw** – Construct instances, list of members (requires Python 3.6)

Example:

```
BitStruct  <-->  Bitwise(Struct(...))

>>> d = BitStruct(
...     "a" / Flag,
...     "b" / Nibble,
...     "c" / BitsInteger(10),
...     "d" / Padding(1),
... )
>>> d.parse(b"\xbe\xef")
```

(continues on next page)

(continued from previous page)

```
Container(a=True)(b=7)(c=887)(d=None)
>>> d.sizeof()
2
```

2.8 Core API: Repeaters

construct.**Array**(*count*, *subcon*, *discard=False*)

Homogenous array of elements, similar to C# generic T[].

Parses into a ListContainer (a list). Parsing and building processes an exact amount of elements. If given list has more or less than count elements, raises RangeError. Size is defined as count multiplied by subcon size, but only if subcon is fixed size.

Operator [] can be used to make Array instances (recommended syntax).

Parameters

- **count** – integer or context lambda, strict amount of elements
- **subcon** – Construct instance, subcon to process individual elements
- **discard** – optional, bool, if set then parsing returns empty list

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- ***RangeError*** – specified count is not valid
- ***RangeError*** – given object has different length than specified count

Can propagate any exception from the lambdas, possibly non-ConstructError.

Example:

```
>>> d = Array(5, Byte) or Byte[5]
>>> d.build(range(5))
b'\x00\x01\x02\x03\x04'
>>> d.parse(_)
[0, 1, 2, 3, 4]
```

construct.**GreedyRange**(*subcon*, *discard=False*)

Homogenous array of elements, similar to C# generic IEnumerable<T>, but works with unknown count of elements by parsing until end of stream.

Parses into a ListContainer (a list). Parsing stops when an exception occured when parsing the subcon, either due to EOF or subcon format not being able to parse the data. Either way, when GreedyRange encounters either failure it seeks the stream back to a position after last successful subcon parsing. Builds from enumerable, each element as-is. Size is undefined.

This class supports stopping. If *StopIf* field is a member, and it evaluates its lambda as positive, this class ends parsing or building as successful without processing further fields.

Parameters

- **subcon** – Construct instance, subcon to process individual elements
- **discard** – optional, bool, if set then parsing returns empty list

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- ***StreamError*** – stream is not seekable and tellable

Can propagate any exception from the lambdas, possibly non-ConstructError.

Example:

```
>>> d = GreedyRange(Byte)
>>> d.build(range(8))
b'\x00\x01\x02\x03\x04\x05\x06\x07'
>>> d.parse(_)
[0, 1, 2, 3, 4, 5, 6, 7]
```

construct.**RepeatUntil**(*predicate*, *subcon*, *discard=False*)

Homogenous array of elements, similar to C# generic IEnumerable<T>, that repeats until the predicate indicates it to stop. Note that the last element (that predicate indicated as True) is included in the return list.

Parse iterates indefinately until last element passed the predicate. Build iterates indefinately over given list, until an element passed the precicate (or raises RepeatError if no element passed it). Size is undefined.

Parameters

- **predicate** – lambda that takes (obj, list, context) and returns True to break or False to continue (or a truthy value)
- **subcon** – Construct instance, subcon used to parse and build each element
- **discard** – optional, bool, if set then parsing returns empty list

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- ***RepeatError*** – consumed all elements in the stream but neither passed the predicate

Can propagate any exception from the lambda, possibly non-ConstructError.

Example:

```
>>> d = RepeatUntil(lambda x,lst,ctx: x > 7, Byte)
>>> d.build(range(20))
b'\x00\x01\x02\x03\x04\x05\x06\x07\x08'
>>> d.parse(b"\x01\xff\x02")
[1, 255]

>>> d = RepeatUntil(lambda x,lst,ctx: lst[-2:] == [0,0], Byte)
>>> d.parse(b"\x01\x00\x00\xff")
[1, 0, 0]
```

2.9 Core API: Special

construct.**Embedded**(*subcon*)

Special wrapper that allows outer multiple-subcons construct to merge fields from another multiple-subcons construct. Embedded does not change a field, only wraps it like a candy with a flag.

> **Warning:** Can only be used between Struct Sequence FocusedSeq Union LazyStruct, although they can be used interchangably, for example Struct can embed fields from a Sequence. There is also *EmbeddedSwitch* macro that pseudo-embeds a Switch. Its not possible to embed IfThenElse.

Parsing building and sizeof are deferred to subcon.

Parameters **subcon** – Construct instance, its fields to embed inside a struct or sequence

Example:

```
>>> outer = Struct(
...     Embedded(Struct(
...         "data" / Bytes(4),
...     )),
... )
>>> outer.parse(b"1234")
Container(data=b'1234')
```

construct.**Renamed**(*subcon*, *newname=None*, *newdocs=None*, *newparsed=None*)

Special wrapper that allows a Struct (or other similar class) to see a field as having a name (or a different name) or having a parsed hook. Library classes do not have names (its None). Renamed does not change a field, only wraps it like a candy with a label. Used internally by / and * operators.

Also this wrapper is responsible for building a path info (a chain of names) that gets attached to error message when parsing, building, or sizeof fails. Fields that are not named do not appear in the path string.

Parsing building and size are deferred to subcon.

Parameters

- **subcon** – Construct instance
- **newname** – optional, string
- **newdocs** – optional, string
- **newparsed** – optional, lambda

Example:

```
>>> "number" / Int32ub
<Renamed: number>
```

2.10 Core API: Miscellaneous

construct.**Const**(*value*, *subcon=None*)

Field enforcing a constant. It is used for file signatures, to validate that the given pattern exists. Data in the stream must strictly match the specified value.

Note that a variable sized subcon may still provide positive verification. Const does not consume a precomputed amount of bytes, but depends on the subcon to read the appropriate amount (eg. VarInt is acceptable). Whatever subcon parses into, gets compared against the specified value.

Parses using subcon and return its value (after checking). Builds using subcon from nothing (or given object, if not None). Size is the same as subcon, unless it raises SizeofError.

Parameters

- **value** – expected value, usually a bytes literal

- **subcon** – optional, Construct instance, subcon used to build value from, assumed to be Bytes if value parameter was a bytes literal

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- ***ConstError*** – parsed data does not match specified value, or building from wrong value
- ***StringError*** – building from non-bytes value, perhaps unicode

Example:

```
>>> d = Const(b"IHDR")
>>> d.build(None)
b'IHDR'
>>> d.parse(b"JPEG")
construct.core.ConstError: expected b'IHDR' but parsed b'JPEG'

>>> d = Const(255, Int32ul)
>>> d.build(None)
b'\xff\x00\x00\x00'
```

construct.**Computed**(*func*)

Field computing a value from the context dictionary or some outer source like os.urandom or random module. Underlying byte stream is unaffected. The source can be non-deterministic.

Parsing and Building return the value returned by the context lambda (although a constant value can also be used). Size is defined as 0 because parsing and building does not consume or produce bytes into the stream.

Parameters **func** – context lambda or constant value

Can propagate any exception from the lambda, possibly non-ConstructError.

Example::

```
>>> d = Struct(
...     "width" / Byte,
...     "height" / Byte,
...     "total" / Computed(this.width * this.height),
... )
>>> d.build(dict(width=4,height=5))
b'\x04\x05'
>>> d.parse(b"12")
Container(width=49, height=50, total=2450)
```

```
>>> d = Computed(7)
>>> d.parse(b"")
7
>>> d = Computed(lambda ctx: 7)
>>> d.parse(b"")
7
```

```
>>> import os
>>> d = Computed(lambda ctx: os.urandom(10))
>>> d.parse(b"")
b'\x98\xc2\xec\x10\x07\xf5\x8e\x98\xc2\xec'
```

construct.**Index**()

Indexes a field inside outer *Array GreedyRange RepeatUntil* context.

Note that you can use this class, or use *this._index* expression instead, depending on how its used. See the examples.

Parsing and building pulls _index key from the context. Size is 0 because stream is unaffected.

Raises ***IndexFieldError*** – did not find either key in context

Example:

```
>>> d = Array(3, Index)
>>> d.parse(b"")
[0, 1, 2]
>>> d = Array(3, Struct("i" / Index))
>>> d.parse(b"")
[Container(i=0), Container(i=1), Container(i=2)]

>>> d = Array(3, Computed(this._index+1))
>>> d.parse(b"")
[1, 2, 3]
>>> d = Array(3, Struct("i" / Computed(this._._index+1)))
>>> d.parse(b"")
[Container(i=1), Container(i=2), Container(i=3)]
```

construct.**Rebuild**(*subcon*, *func*)

Field where building does not require a value, because the value gets recomputed when needed. Comes handy when building a Struct from a dict with missing keys. Useful for length and count fields when *Prefixed* and *PrefixedArray* cannot be used.

Parsing defers to subcon. Building is defered to subcon, but it builds from a value provided by the context lambda (or constant). Size is the same as subcon, unless it raises SizeofError.

Difference between Default and Rebuild, is that in first the build value is optional and in second the build value is ignored.

Parameters

- **subcon** – Construct instance
- **func** – context lambda or constant value

Raises ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes

Can propagate any exception from the lambda, possibly non-ConstructError.

Example:

```
>>> d = Struct(
...     "count" / Rebuild(Byte, len_(this.items)),
...     "items" / Byte[this.count],
... )
>>> d.build(dict(items=[1,2,3]))
b'\x03\x01\x02\x03'
```

construct.**Default**(*subcon*, *value*)

Field where building does not require a value, because the value gets taken from default. Comes handy when building a Struct from a dict with missing keys.

Parsing defers to subcon. Building is defered to subcon, but it builds from a default (if given object is None) or from given object. Building does not require a value, but can accept one. Size is the same as subcon, unless it raises SizeofError.

Difference between Default and Rebuild, is that in first the build value is optional and in second the build value is ignored.

Parameters

- **subcon** – Construct instance
- **value** – context lambda or constant value

Raises ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes

Can propagate any exception from the lambda, possibly non-ConstructError.

Example:

```
>>> d = Struct(
...     "a" / Default(Byte, 0),
... )
>>> d.build(dict(a=1))
b'\x01'
>>> d.build(dict())
b'\x00'
```

construct.**Check**(*func*)

Checks for a condition, and raises CheckError if the check fails.

Parsing and building return nothing (but check the condition). Size is 0 because stream is unaffected.

Parameters **func** – bool or context lambda, that gets run on parsing and building

Raises ***CheckError*** – lambda returned false

Can propagate any exception from the lambda, possibly non-ConstructError.

Example:

```
Check(lambda ctx: len(ctx.payload.data) == ctx.payload_len)
Check(len_(this.payload.data) == this.payload_len)
```

construct.**Error**()

Raises ExplicitError, unconditionally.

Parsing and building always raise ExplicitError. Size is undefined.

Raises ***ExplicitError*** – unconditionally, on parsing and building

Example:

```
>>> d = Struct("num"/Byte, Error)
>>> d.parse(b"data...")
construct.core.ExplicitError: Error field was activated during parsing
```

construct.**FocusedSeq**(*parsebuildfrom*, **subcons*, ***subconskw*)

Allows constructing more elaborate "adapters" than Adapter class.

Parse does parse all subcons in sequence, but returns only the element that was selected (discards other values). Build does build all subcons in sequence, where each gets build from nothing (except the selected subcon which is given the object). Size is the sum of all subcon sizes, unless any subcon raises SizeofError.

This class does context nesting, meaning its members are given access to a new dictionary where the "_" entry points to the outer context. When parsing, each member gets parsed and subcon parse return value is inserted into context under matching key only if the member was named. When building, the matching entry gets inserted

into context before subcon gets build, and if subcon build returns a new value (not None) that gets replaced in the context.

This class supports embedding. *Embedded* semantics dictate, that during instance creation (in ctor), each field is checked for embedded flag, and its subcon members are merged. This changes behavior of some code examples. Only few classes are supported: Struct Sequence FocusedSeq Union LazyStruct, although those can be used interchangably (a Struct can embed a Sequence, or rather its members).

This class exposes subcons as attributes. You can refer to subcons that were inlined (and therefore do not exist as variable in the namespace) by accessing the struct attributes, under same name. Also note that compiler does not support this feature. See examples.

This class exposes subcons in the context. You can refer to subcons that were inlined (and therefore do not exist as variable in the namespace) within other inlined fields using the context. Note that you need to use a lambda (*this* expression is not supported). Also note that compiler does not support this feature. See examples.

This class is used internally to implement *PrefixedArray*.

Parameters

- **parsebuildfrom** – string name or context lambda, selects a subcon
- ***subcons** – Construct instances, list of members, some can be named
- ****subconskw** – Construct instances, list of members (requires Python 3.6)

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- **UnboundLocalError** – selector does not match any subcon

Can propagate any exception from the lambda, possibly non-ConstructError.

Excample:

```
>>> d = FocusedSeq("num", Const(b"SIG"), "num"/Byte, Terminated)
>>> d.parse(b"SIG\xff")
255
>>> d.build(255)
b'SIG\xff'

>>> d = FocusedSeq("animal",
...     "animal" / Enum(Byte, giraffe=1),
... )
>>> d.animal.giraffe
'giraffe'
>>> d = FocusedSeq("count",
...     "count" / Byte,
...     "data" / Padding(lambda this: this.count - this._subcons.count.sizeof()),
... )
>>> d.build(4)
b'\x04\x00\x00\x00'

PrefixedArray <--> FocusedSeq("items",
    "count" / Rebuild(lengthfield, len_(this.items)),
    "items" / subcon[this.count],
)
```

construct.**Pickled**()

Preserves arbitrary Python objects.

Parses using pickle.load() and builds using pickle.dump() functions, using default Pickle binary protocol. Size is undefined.

Raises ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes

Can propagate pickle.load() and pickle.dump() exceptions.

Example:

```
>>> x = [1, 2.3, {}]
>>> Pickled.build(x)
b'\x80\x03]q\x00(K\x01G@\x02ffffff}q\x01e.'
>>> Pickled.parse(_)
[1, 2.3, {}]
```

construct.**Numpy**()

Preserves numpy arrays (both shape, dtype and values).

Parses using numpy.load() and builds using numpy.save() functions, using Numpy binary protocol. Size is undefined.

Raises

- **ImportError** – numpy could not be imported during parsing or building
- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes

Can propagate numpy.load() and numpy.save() exceptions.

Example:

```
>>> import numpy
>>> a = numpy.asarray([1,2,3])
>>> Numpy.build(a)
b"\x93NUMPY\x01\x00F\x00{'descr': '<i8', 'fortran_order': False, 'shape': (3,), }␣
↪            ␣
↪\n\x01\x00\x00\x00\x00\x00\x00\x00\x02\x00\x00\x00\x00\x00\x00\x00\x03\x00\x00\x00\x00\x00\x00
↪"
>>> Numpy.parse(_)
array([1, 2, 3])
```

construct.**NamedTuple**(*tuplename*, *tuplefields*, *subcon*)

Both arrays, structs, and sequences can be mapped to a namedtuple from collections module. To create a named tuple, you need to provide a name and a sequence of fields, either a string with space-separated names or a list of string names, like the standard namedtuple.

Parses into a collections.namedtuple instance, and builds from such instance (although it also builds from lists and dicts). Size is undefined.

Parameters

- **tuplename** – string
- **tuplefields** – string or list of strings
- **subcon** – Construct instance, either Struct Sequence Array GreedyRange

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes

- ***NamedTupleError*** – subcon is neither Struct Sequence Array GreedyRange

Can propagate collections exceptions.

Example:

```
>>> d = NamedTuple("coord", "x y z", Byte[3])
>>> d = NamedTuple("coord", "x y z", Byte >> Byte >> Byte)
>>> d = NamedTuple("coord", "x y z", "x"/Byte + "y"/Byte + "z"/Byte)
>>> d.parse(b"123")
coord(x=49, y=50, z=51)
```

construct.**Timestamp**(*subcon*, *unit*, *epoch*)

Datetime, represented as Arrow object.

Note that accuracy is not guaranteed, because building rounds the value to integer (even when Float subcon is used), due to floating-point errors in general, and because MSDOS scheme has only 5-bit (32 values) seconds field (seconds are rounded to multiple of 2).

Unit is a fraction of a second. 1 is second resolution, 10**-3 is milliseconds resolution, 10**-6 is microseconds resolution, etc. Usually its 1 on Unix and MacOSX, 10**-7 on Windows. Epoch is a year (if integer) or a specific day (if Arrow object). Usually its 1970 on Unix, 1904 on MacOSX, 1600 on Windows. MSDOS format doesnt support custom unit or epoch, it uses 2-seconds resolution and 1980 epoch.

Parameters

- **subcon** – Construct instance like Int* Float*, or Int32ub with msdos format
- **unit** – integer or float, or msdos string
- **epoch** – integer, or Arrow instance, or msdos string

Raises

- **ImportError** – arrow could not be imported during ctor
- ***TimestampError*** – subcon is not a Construct instance
- ***TimestampError*** – unit or epoch is a wrong type

Example:

```
>>> d = Timestamp(Int64ub, 1., 1970)
>>> d.parse(b'\x00\x00\x00\x00ZIz\x00')
<Arrow [2018-01-01T00:00:00+00:00]>
>>> d = Timestamp(Int32ub, "msdos", "msdos")
>>> d.parse(b'H9\x8c"')
<Arrow [2016-01-25T17:33:04+00:00]>
```

construct.**Hex**(*subcon*)

Adapter for displaying hexadecimal/hexlified representation of integers/bytes/RawCopy dictionaries.

Parsing results in int-alike bytes-alike or dict-alike object, whose only difference from original is pretty-printing. If you look at the result, you will be presented with its *repr* which remains as-is. If you print it, then you will see its *str* whic is a hexlified representation. Building and sizeof defer to subcon.

To obtain a hexlified string (like before Hex HexDump changed semantics) use binascii.(un)hexlify on parsed results.

Example:

```
>>> d = Hex(Int32ub)
>>> obj = d.parse(b"\x00\x00\x01\x02")
>>> obj
258
>>> print(obj)
0x00000102

>>> d = Hex(GreedyBytes)
>>> obj = d.parse(b"\x00\x00\x01\x02")
>>> obj
b'\x00\x00\x01\x02'
>>> print(obj)
unhexlify('00000102')

>>> d = Hex(RawCopy(Int32ub))
>>> obj = d.parse(b"\x00\x00\x01\x02")
>>> obj
{'data': b'\x00\x00\x01\x02',
 'length': 4,
 'offset1': 0,
 'offset2': 4,
 'value': 258}
>>> print(obj)
unhexlify('00000102')
```

construct.**HexDump**(*subcon*)

Adapter for displaying hexlified representation of bytes/RawCopy dictionaries.

Parsing results in bytes-alike or dict-alike object, whose only difference from original is pretty-printing. If you look at the result, you will be presented with its *repr* which remains as-is. If you print it, then you will see its *str* whic is a hexlified representation. Building and sizeof defer to subcon.

To obtain a hexlified string (like before Hex HexDump changed semantics) use construct.lib.hexdump on parsed results.

Example:

```
>>> d = HexDump(GreedyBytes)
>>> obj = d.parse(b"\x00\x00\x01\x02")
>>> obj
b'\x00\x00\x01\x02'
>>> print(obj)
hexundump('''
0000   00 00 01 02                                       ....
''')

>>> d = HexDump(RawCopy(Int32ub))
>>> obj = d.parse(b"\x00\x00\x01\x02")
>>> obj
{'data': b'\x00\x00\x01\x02',
 'length': 4,
 'offset1': 0,
 'offset2': 4,
 'value': 258}
>>> print(obj)
hexundump('''
0000   00 00 01 02                                       ....
''')
```

2.11 Core API: Conditional

construct.**Union**(*parsefrom*, **subcons*, ***subconskw*)

Treats the same data as multiple constructs (similar to C union) so you can look at the data in multiple views. Fields are usually named (so parsed values are inserted into dictionary under same name). *Embedded* fields do not need to (and should not) be named.

Parses subcons in sequence, and reverts the stream back to original position after each subcon. Afterwards, advances the stream by selected subcon. Builds from first subcon that has a matching key in given dict. Size is undefined (because parsefrom is not used for building).

This class does context nesting, meaning its members are given access to a new dictionary where the "_" entry points to the outer context. When parsing, each member gets parsed and subcon parse return value is inserted into context under matching key only if the member was named. When building, the matching entry gets inserted into context before subcon gets build, and if subcon build returns a new value (not None) that gets replaced in the context.

This class supports embedding. *Embedded* semantics dictate, that during instance creation (in ctor), each field is checked for embedded flag, and its subcon members are merged. This changes behavior of some code examples. Only few classes are supported: Struct Sequence FocusedSeq Union LazyStruct, although those can be used interchangably (a Struct can embed a Sequence, or rather its members).

This class exposes subcons as attributes. You can refer to subcons that were inlined (and therefore do not exist as variable in the namespace) by accessing the struct attributes, under same name. Also note that compiler does not support this feature. See examples.

This class exposes subcons in the context. You can refer to subcons that were inlined (and therefore do not exist as variable in the namespace) within other inlined fields using the context. Note that you need to use a lambda (*this* expression is not supported). Also note that compiler does not support this feature. See examples.

> **Warning:** If you skip *parsefrom* parameter then stream will be left back at starting offset, not seeked to any common denominator.

Parameters

- **parsefrom** – how to leave stream after parsing, can be integer index or string name selecting a subcon, or None (leaves stream at initial offset, the default), or context lambda
- ***subcons** – Construct instances, list of members, some can be anonymous
- ****subconskw** – Construct instances, list of members (requires Python 3.6)

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- ***StreamError*** – stream is not seekable and tellable
- ***UnionError*** – selector does not match any subcon, or dict given to build does not contain any keys matching any subcon
- **IndexError** – selector does not match any subcon
- **KeyError** – selector does not match any subcon

Can propagate any exception from the lambda, possibly non-ConstructError.

Example:

```
>>> d = Union(0,
...     "raw" / Bytes(8),
...     "ints" / Int32ub[2],
...     "shorts" / Int16ub[4],
...     "chars" / Byte[8],
... )
>>> d.parse(b"12345678")
Container(raw=b'12345678', ints=[825373492, 892745528], shorts=[12594, 13108,␣
↪13622, 14136], chars=[49, 50, 51, 52, 53, 54, 55, 56])
>>> d.build(dict(chars=range(8)))
b'\x00\x01\x02\x03\x04\x05\x06\x07'

>>> d = Union(None,
...     "animal" / Enum(Byte, giraffe=1),
... )
>>> d.animal.giraffe
'giraffe'
>>> d = Union(None,
...     "chars" / Byte[4],
...     "data" / Bytes(lambda this: this._subcons.chars.sizeof()),
... )
>>> d.parse(b"\x01\x02\x03\x04")
Container(chars=[1, 2, 3, 4])(data=b'\x01\x02\x03\x04')

Alternative syntax, but requires Python 3.6 or any PyPy:
>>> Union(0, raw=Bytes(8), ints=Int32ub[2], shorts=Int16ub[4], chars=Byte[8])
```

construct.**Select**(**subcons*, ***subconskw*)

Selects the first matching subconstruct.

Parses and builds by literally trying each subcon in sequence until one of them parses or builds without exception. Stream gets reverted back to original position after each failed attempt, but not if parsing succeeds. Size is not defined.

Parameters

- ***subcons** – Construct instances, list of members, some can be anonymous
- ****subconskw** – Construct instances, list of members (requires Python 3.6)

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- ***StreamError*** – stream is not seekable and tellable
- ***SelectError*** – neither subcon succeded when parsing or building

Example:

```
>>> d = Select(Int32ub, CString("utf8"))
>>> d.build(1)
b'\x00\x00\x00\x01'
>>> d.build(u"")
b'\xd0\x90\xd1\x84\xd0\xbe\xd0\xbd\x00'

Alternative syntax, but requires Python 3.6 or any PyPy:
>>> Select(num=Int32ub, text=CString("utf8"))
```

construct.**Optional**(*subcon*)
Makes an optional field.

Parsing attempts to parse subcon. If sub-parsing fails, returns None and reports success. Building attempts to build subcon. If sub-building fails, writes nothing and reports success. Size is undefined, because whether bytes would be consumed or produced depends on actual data and actual context.

Parameters **subcon** – Construct instance

Example:

```
Optional  <-->  Select(subcon, Pass)

>>> d = Optional(Int64ul)
>>> d.parse(b"12345678")
4050765991979987505
>>> d.parse(b"")
None
>>> d.build(1)
b'\x01\x00\x00\x00\x00\x00\x00\x00'
>>> d.build(None)
b''
```

construct.**If**(*condfunc*, *subcon*)
If-then conditional construct.

Parsing evaluates condition, if True then subcon is parsed, otherwise just returns None. Building also evaluates condition, if True then subcon gets build from, otherwise does nothing. Size is either same as subcon or 0, depending how condfunc evaluates.

Parameters

- **condfunc** – bool or context lambda (or a truthy value)
- **subcon** – Construct instance, used if condition indicates True

Raises ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes

Can propagate any exception from the lambda, possibly non-ConstructError.

Example:

```
If <--> IfThenElse(condfunc, subcon, Pass)

>>> d = If(this.x > 0, Byte)
>>> d.build(255, x=1)
b'\xff'
>>> d.build(255, x=0)
b''
```

construct.**IfThenElse**(*condfunc*, *thensubcon*, *elsesubcon*)
If-then-else conditional construct, similar to ternary operator.

Parsing and building evaluates condition, and defers to either subcon depending on the value. Size is computed the same way.

Parameters

- **condfunc** – bool or context lambda (or a truthy value)
- **thensubcon** – Construct instance, used if condition indicates True

- **elsesubcon** – Construct instance, used if condition indicates False

Raises ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes

Can propagate any exception from the lambda, possibly non-ConstructError.

Example:

```
>>> d = IfThenElse(this.x > 0, VarInt, Byte)
>>> d.build(255, dict(x=1))
b'\xff\x01'
>>> d.build(255, dict(x=0))
b'\xff'
```

construct.**Switch**(*keyfunc*, *cases*, *default=None*)

A conditional branch.

Parsing and building evaluate keyfunc and select a subcon based on the value and dictionary entries. Dictionary (cases) maps values into subcons. If no case matches then *default* is used (that is Pass by default). Note that *default* is a Construct instance, not a dictionary key. Size is evaluated in same way as parsing and building, by evaluating keyfunc and selecting a field accordingly.

Parameters

- **keyfunc** – context lambda or constant, that matches some key in cases
- **cases** – dict mapping keys to Construct instances
- **default** – optional, Construct instance, used when keyfunc is not found in cases, Pass is default value for this parameter, Error is a possible value for this parameter

Raises ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes

Can propagate any exception from the lambda, possibly non-ConstructError.

Example:

```
>>> d = Switch(this.n, { 1:Int8ub, 2:Int16ub, 4:Int32ub })
>>> d.build(5, n=1)
b'\x05'
>>> d.build(5, n=4)
b'\x00\x00\x00\x05'

>>> d = Switch(this.n, {}, default=Byte)
>>> d.parse(b"\x01", n=255)
1
>>> d.build(1, n=255)
b"\x01"
```

construct.**EmbeddedSwitch**(*merged*, *selector*, *mapping*)

Macro that simulates embedding Switch, which under new embedding semantics is not possible. This macro does NOT produce a Switch. It generates classes that behave the same way as you would expect from embedded Switch, only that. Instance created by this macro CAN be embedded.

Both *merged* and all values in *mapping* must be Struct instances. Macro re-creates a single struct that contains all fields, where each field is wrapped in *If(selector == key, ...)*. Note that resulting dictionary contains None values for fields that would not be chosen by switch. Note also that if selector does not match any cases, it passes successfully (default Switch behavior).

All fields should have unique names. Otherwise fields that were not selected during parsing may return None and override other fields context entries that have same name. This is because *If* field returns None value if condition is not met, but the Struct inserts that None value into the context entry regardless.

Parameters

- **merged** – Struct instance
- **selector** – this expression, that references one of *merged* fields
- **mapping** – dict with values being Struct instances

Example:

```
d = EmbeddedSwitch(
    Struct(
        "type" / Byte,
    ),
    this.type,
    {
        0: Struct("name" / PascalString(Byte, "utf8")),
        1: Struct("value" / Byte),
    }
)

# generates essentially following
d = Struct(
    "type" / Byte,
    "name" / If(this.type == 0, PascalString(Byte, "utf8")),
    "value" / If(this.type == 1, Byte),
)

# both parse like following
>>> d.parse(b"\x00\x00")
Container(type=0, name=u'', value=None)
>>> d.parse(b"\x01\x00")
Container(type=1, name=None, value=0)
```

construct.**StopIf**(*condfunc*)

Checks for a condition, and stops certain classes (*Struct* *Sequence* *GreedyRange*) from parsing or building further.

Parsing and building check the condition, and raise StopFieldError if indicated. Size is undefined.

Parameters **condfunc** – bool or context lambda (or truthy value)

Raises ***StopFieldError*** – used internally

Can propagate any exception from the lambda, possibly non-ConstructError.

Example:

```
>>> Struct('x'/Byte, StopIf(this.x == 0), 'y'/Byte)
>>> Sequence('x'/Byte, StopIf(this.x == 0), 'y'/Byte)
>>> GreedyRange(FocusedSeq(0, 'x'/Byte, StopIf(this.x == 0)))
```

2.12 Core API: Alignment and Padding

construct.**Padding**(*length*, *pattern='\x00'*)

Appends null bytes.

Parsing consumes specified amount of bytes and discards it. Building writes specified pattern byte multiplied into specified length. Size is same as specified.

Parameters

- **length** – integer or context lambda, length of the padding
- **pattern** – b-character, padding pattern, default is \x00

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- ***PaddingError*** – length was negative
- ***PaddingError*** – pattern was not bytes (b-character)

Can propagate any exception from the lambda, possibly non-ConstructError.

Example:

```
>>> d = Padding(4) or Padded(4, Pass)
>>> d.build(None)
b'\x00\x00\x00\x00'
>>> d.parse(b"****")
None
>>> d.sizeof()
4
```

construct.**Padded**(*length*, *subcon*, *pattern='\x00'*)

Appends additional null bytes to achieve a length.

Parsing first parses the subcon, then uses stream.tell() to measure how many bytes were read and consumes additional bytes accordingly. Building first builds the subcon, then uses stream.tell() to measure how many bytes were written and produces additional bytes accordingly. Size is same as *length*, but negative amount results in error. Note that subcon can actually be variable size, it is the eventual amount of bytes that is read or written during parsing or building that determines actual padding.

Parameters

- **length** – integer or context lambda, length of the padding
- **subcon** – Construct instance
- **pattern** – optional, b-character, padding pattern, default is \x00

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- ***PaddingError*** – length is negative
- ***PaddingError*** – subcon read or written more than the length (would cause negative pad)
- ***PaddingError*** – pattern is not bytes of length 1

Can propagate any exception from the lambda, possibly non-ConstructError.

Example:

```
>>> d = Padded(4, Byte)
>>> d.build(255)
b'\xff\x00\x00\x00'
>>> d.parse(_)
255
>>> d.sizeof()
4

>>> d = Padded(4, VarInt)
>>> d.build(1)
b'\x01\x00\x00\x00'
>>> d.build(70000)
b'\xf0\xa2\x04\x00'
```

construct.**Aligned**(*modulus*, *subcon*, *pattern='\x00'*)

Appends additional null bytes to achieve a length that is shortest multiple of a modulus.

Note that subcon can actually be variable size, it is the eventual amount of bytes that is read or written during parsing or building that determines actual padding.

Parsing first parses subcon, then consumes an amount of bytes to sum up to specified length, and discards it. Building first builds subcon, then writes specified pattern byte to sum up to specified length. Size is subcon size plus modulo remainder, unless SizeofError was raised.

Parameters

- **modulus** – integer or context lambda, modulus to final length
- **subcon** – Construct instance
- **pattern** – optional, b-character, padding pattern, default is \x00

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- ***PaddingError*** – modulus was less than 2
- ***PaddingError*** – pattern was not bytes (b-character)

Can propagate any exception from the lambda, possibly non-ConstructError.

Example:

```
>>> d = Aligned(4, Int16ub)
>>> d.parse(b'\x00\x01\x00\x00')
1
>>> d.sizeof()
4
```

construct.**AlignedStruct**(*modulus*, **subcons*, ***subconskw*)

Makes a structure where each field is aligned to the same modulus (it is a struct of aligned fields, NOT an aligned struct).

See *Aligned* and *Struct* for semantics and raisable exceptions.

Parameters

- **modulus** – integer or context lambda, passed to each member

- ***subcons** – Construct instances, list of members, some can be anonymous
- ****subconskw** – Construct instances, list of members (requires Python 3.6)

Example:

```
>>> d = AlignedStruct(4, "a"/Int8ub, "b"/Int16ub)
>>> d.build(dict(a=0xFF,b=0xFFFF))
b'\xff\x00\x00\x00\xff\xff\x00\x00'
```

2.13 Core API: Streaming

construct.**Pointer**(*offset*, *subcon*, *stream=None*)

Jumps in the stream forth and back for one field.

Parsing and building seeks the stream to new location, processes subcon, and seeks back to original location. Size is defined as 0 but that does not mean no bytes are written into the stream.

Offset can be positive, indicating a position from stream beginning forward, or negative, indicating a position from EOF backwards.

Parameters

- **offset** – integer or context lambda, positive or negative
- **subcon** – Construct instance
- **stream** – None to use original stream (default), or context lambda to provide a different stream

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- ***StreamError*** – stream is not seekable and tellable

Can propagate any exception from the lambda, possibly non-ConstructError.

Example:

```
>>> d = Pointer(8, Bytes(1))
>>> d.parse(b"abcdefghijkl")
b'i'
>>> d.build(b"Z")
b'\x00\x00\x00\x00\x00\x00\x00\x00Z'
```

construct.**Peek**(*subcon*)

Peeks at the stream.

Parsing sub-parses (and returns None if failed), then reverts stream to original position. Building does nothing (its NOT deferred). Size is defined as 0 because there is no building.

This class is used in *Union* class to parse each member.

Parameters **subcon** – Construct instance

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes

- ***StreamError*** – stream is not seekable and tellable

Example:

```
>>> d = Sequence(Peek(Int8ub), Peek(Int16ub))
>>> d.parse(b"\x01\x02")
[1, 258]
>>> d.sizeof()
0
```

construct.**Seek**(*at*, *whence=0*)

Seeks the stream.

Parsing and building seek the stream to given location (and whence), and return stream.seek() return value. Size is not defined.

See also:

Analog *Pointer* wrapper that has same side effect but also processes a subcon, and also seeks back.

Parameters

- **at** – integer or context lambda, where to jump to
- **whence** – optional, integer or context lambda, is the offset from beginning (0) or from current position (1) or from EOF (2), default is 0

Raises ***StreamError*** – stream is not seekable

Can propagate any exception from the lambda, possibly non-ConstructError.

Example:

```
>>> d = (Seek(5) >> Byte)
>>> d.parse(b"01234x")
[5, 120]

>>> d = (Bytes(10) >> Seek(5) >> Byte)
>>> d.build([b"0123456789", None, 255])
b'01234\xff6789'
```

construct.**Tell**()

Tells the stream.

Parsing and building return current stream offset using using stream.tell(). Size is defined as 0 because parsing and building does not consume or add into the stream.

Tell is useful for adjusting relative offsets to absolute positions, or to measure sizes of Constructs. To get an absolute pointer, use a Tell plus a relative offset. To get a size, place two Tells and measure their difference using a Compute field. However, its recommended to use *RawCopy* instead of manually extracting two positions and computing difference.

Raises ***StreamError*** – stream is not tellable

Example:

```
>>> d = Struct("num"/VarInt, "offset"/Tell)
>>> d.parse(b"X")
Container(num=88)(offset=1)
>>> d.build(dict(num=88))
b'X'
```

construct.**Pass**()
No-op construct, useful as default cases for Switch and Enum.

Parsing returns None. Building does nothing. Size is 0 by definition.

Example:

```
>>> Pass.parse(b"")
None
>>> Pass.build(None)
b''
>>> Pass.sizeof()
0
```

construct.**Terminated**()
Asserts end of stream (EOF). You can use it to ensure no more unparsed data follows in the stream.

Parsing checks if stream reached EOF, and raises TerminatedError if not. Building does nothing. Size is defined as 0 because parsing and building does not consume or add into the stream, as far as other constructs see it.

Raises ***TerminatedError*** – stream not at EOF when parsing

Example:

```
>>> Terminated.parse(b"")
None
>>> Terminated.parse(b"remaining")
construct.core.TerminatedError: expected end of stream
```

2.14 Core API: Tunneling

construct.**RawCopy**(*subcon*)
Used to obtain byte representation of a field (aside of object value).

Returns a dict containing both parsed subcon value, the raw bytes that were consumed by subcon, starting and ending offset in the stream, and amount in bytes. Builds either from raw bytes representation or a value used by subcon. Size is same as subcon.

Object is a dictionary with either "data" or "value" keys, or both.

Parameters **subcon** – Construct instance

Raises

- ***StreamError*** – stream is not seekable and tellable
- ***RawCopyError*** – building and neither data or value was given
- ***StringError*** – building from non-bytes value, perhaps unicode

Example:

```
>>> d = RawCopy(Byte)
>>> d.parse(b"\xff")
Container(data=b'\xff')(value=255)(offset1=0)(offset2=1)(length=1)
>>> d.build(dict(data=b"\xff"))
'\xff'
>>> d.build(dict(value=255))
'\xff'
```

construct.**ByteSwapped**(*subcon*)

Swaps the byte order within boundaries of given subcon. Requires a fixed sized subcon.

Parameters **subcon** – Construct instance, subcon on top of byte swapped bytes

Raises ***SizeofError*** – ctor or compiler could not compute subcon size

See *Transformed* and *Restreamed* for raisable exceptions.

Example:

```
Int24ul <--> ByteSwapped(Int24ub) <--> BytesInteger(3, swapped=True) <-->␣
↪ByteSwapped(BytesInteger(3))
```

construct.**BitsSwapped**(*subcon*)

Swaps the bit order within each byte within boundaries of given subcon. Does NOT require a fixed sized subcon.

Parameters **subcon** – Construct instance, subcon on top of bit swapped bytes

Raises ***SizeofError*** – compiler could not compute subcon size

See *Transformed* and *Restreamed* for raisable exceptions.

Example:

```
>>> d = Bitwise(Bytes(8))
>>> d.parse(b"\x01")
'\x00\x00\x00\x00\x00\x00\x00\x01'
>>>> BitsSwapped(d).parse(b"\x01")
'\x01\x00\x00\x00\x00\x00\x00\x00'
```

construct.**Prefixed**(*lengthfield*, *subcon*, *includelength=False*)

Prefixes a field with byte count.

Parses the length field. Then reads that amount of bytes, and parses subcon using only those bytes. Constructs that consume entire remaining stream are constrained to consuming only the specified amount of bytes (a substream). When building, data gets prefixed by its length. Optionally, length field can include its own size. Size is the sum of both fields sizes, unless either raises SizeofError.

Analog to *PrefixedArray* which prefixes with an element count, instead of byte count. Semantics is similar but implementation is different.

VarInt is recommended for new protocols, as it is more compact and never overflows.

Parameters

- **lengthfield** – Construct instance, field used for storing the length
- **subcon** – Construct instance, subcon used for storing the value
- **includelength** – optional, bool, whether length field should include its own size, default is False

Raises ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes

Example:

```
>>> d = Prefixed(VarInt, GreedyRange(Int32ul))
>>> d.parse(b"\x08abcdefgh")
[1684234849, 1751606885]

>>> d = PrefixedArray(VarInt, Int32ul)
```

(continues on next page)

(continued from previous page)

```
>>> d.parse(b"\x02abcdefgh")
[1684234849, 1751606885]
```

construct.**PrefixedArray**(*countfield*, *subcon*)

Prefixes an array with item count (as opposed to prefixed by byte count, see *Prefixed*).

VarInt is recommended for new protocols, as it is more compact and never overflows.

Parameters

- **countfield** – Construct instance, field used for storing the element count
- **subcon** – Construct instance, subcon used for storing each element

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- ***RangeError*** – consumed or produced too little elements

Example:

```
>>> d = Prefixed(VarInt, GreedyRange(Int32ul))
>>> d.parse(b"\x08abcdefgh")
[1684234849, 1751606885]

>>> d = PrefixedArray(VarInt, Int32ul)
>>> d.parse(b"\x02abcdefgh")
[1684234849, 1751606885]
```

construct.**FixedSized**(*length*, *subcon*)

Restricts parsing to specified amount of bytes.

Parsing reads *length* bytes, then defers to subcon using new BytesIO with said bytes. Building builds the subcon using new BytesIO, then writes said data and additional null bytes accordingly. Size is same as *length*, although negative amount raises an error as well.

Parameters

- **length** – integer or context lambda, total amount of bytes (both data and padding)
- **subcon** – Construct instance

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- ***PaddingError*** – length is negative
- ***PaddingError*** – subcon written more bytes than entire length (negative padding)

Can propagate any exception from the lambda, possibly non-ConstructError.

Example:

```
>>> d = FixedSized(10, Byte)
>>> d.parse(b'\xff\x00\x00\x00\x00\x00\x00\x00\x00\x00')
255
>>> d.build(255)
b'\xff\x00\x00\x00\x00\x00\x00\x00\x00\x00'
```

(continues on next page)

(continued from previous page)

```
>>> d.sizeof()
10
```

construct.**NullTerminated**(*subcon*, *term='\x00'*, *include=False*, *consume=True*, *require=True*)

Restricts parsing to bytes preceding a null byte.

Parsing reads one byte at a time and accumulates it with previous bytes. When term was found, (by default) consumes but discards the term. When EOF was found, (by default) raises same StreamError exception. Then subcon is parsed using new BytesIO made with said data. Building builds the subcon and then writes the term. Size is undefined.

The term can be multiple bytes, to support string classes with UTF16/32 encodings.

Parameters

- **subcon** – Construct instance
- **term** – optional, bytes, terminator byte-string, default is x00 single null byte
- **include** – optional, bool, if to include terminator in resulting data, default is False
- **consume** – optional, bool, if to consume terminator or leave it in the stream, default is True
- **require** – optional, bool, if EOF results in failure or not, default is True

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- ***StreamError*** – encountered EOF but require is not disabled
- ***PaddingError*** – terminator is less than 1 bytes in length

Example:

```
>>> d = NullTerminated(Byte)
>>> d.parse(b'\xff\x00')
255
>>> d.build(255)
b'\xff\x00'
```

construct.**NullStripped**(*subcon*, *pad='\x00'*)

Restricts parsing to bytes except padding left of EOF.

Parsing reads entire stream, then strips the data from right to left of null bytes, then parses subcon using new BytesIO made of said data. Building defers to subcon as-is. Size is undefined, because it reads till EOF.

The pad can be multiple bytes, to support string classes with UTF16/32 encodings.

Parameters

- **subcon** – Construct instance
- **pad** – optional, bytes, padding byte-string, default is x00 single null byte

Raises ***PaddingError*** – pad is less than 1 bytes in length

Example:

```
>>> d = NullStripped(Byte)
>>> d.parse(b'\xff\x00\x00')
255
```

(continues on next page)

(continued from previous page)

```
>>> d.build(255)
b'\xff'
```

construct.**RestreamData**(*datafunc*, *subcon*)

Parses a field on external data (but does not build).

Parsing defers to subcon, but provides it a separate BytesIO stream based on data provided by datafunc (a bytes literal or another BytesIO stream or Construct instances that returns bytes or context lambda). Building does nothing. Size is 0 because as far as other fields see it, this field does not produce or consume any bytes from the stream.

Parameters

- **datafunc** – bytes or BytesIO or Construct instance (that parses into bytes) or context lambda, provides data for subcon to parse from
- **subcon** – Construct instance

Can propagate any exception from the lambdas, possibly non-ConstructError.

Example:

```
>>> d = RestreamData(b"\x01", Int8ub)
>>> d.parse(b"")
1
>>> d.build(0)
b''

>>> d = RestreamData(NullTerminated(GreedyBytes), Int16ub)
>>> d.parse(b"\x01\x02\x00")
0x0102
>>> d = RestreamData(FixedSized(2, GreedyBytes), Int16ub)
>>> d.parse(b"\x01\x02\x00")
0x0102
```

construct.**Transformed**(*subcon*, *decodefunc*, *decodeamount*, *encodefunc*, *encodeamount*)

Transforms bytes between the underlying stream and the (fixed-sized) subcon.

Parsing reads a specified amount (or till EOF), processes data using a bytes-to-bytes decoding function, then parses subcon using those data. Building does build subcon into separate bytes, then processes it using encoding bytes-to-bytes function, then writes those data into main stream. Size is reported as *decodeamount* or *encodeamount* if those are equal, otherwise its SizeofError.

Used internally to implement *Bitwise* *Bytewise* *ByteSwapped* *BitsSwapped*.

Possible use-cases include encryption, obfuscation, byte-level encoding.

Warning: Remember that subcon must consume (or produce) an amount of bytes that is same as *decodeamount* (or *encodeamount*).

Warning: Do NOT use seeking/telling classes inside Transformed context.

Parameters

- **subcon** – Construct instance

- **decodefunc** – bytes-to-bytes function, applied before parsing subcon
- **decodeamount** – integer, amount of bytes to read
- **encodefunc** – bytes-to-bytes function, applied after building subcon
- **encodeamount** – integer, amount of bytes to write

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- ***StreamError*** – subcon build and encoder transformed more or less than *encodeamount* bytes, if amount is specified
- ***StringError*** – building from non-bytes value, perhaps unicode

Can propagate any exception from the lambdas, possibly non-ConstructError.

Example:

```
>>> d = Transformed(Bytes(16), bytes2bits, 2, bits2bytes, 2)
>>> d.parse(b"\x00\x00")
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'

>>> d = Transformed(GreedyBytes, bytes2bits, None, bits2bytes, None)
>>> d.parse(b"\x00\x00")
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
```

construct.**Restreamed**(*subcon*, *decoder*, *decoderunit*, *encoder*, *encoderunit*, *sizecomputer*)

Transforms bytes between the underlying stream and the (variable-sized) subcon.

Used internally to implement *Bitwise* *Bytewise* *ByteSwapped* *BitsSwapped*.

Warning: Remember that subcon must consume or produce an amount of bytes that is a multiple of encoding or decoding units. For example, in a Bitwise context you should process a multiple of 8 bits or the stream will fail during parsing/building.

Warning: Do NOT use seeking/telling classes inside Restreamed context.

Parameters

- **subcon** – Construct instance
- **decoder** – bytes-to-bytes function, used on data chunks when parsing
- **decoderunit** – integer, decoder takes chunks of this size
- **encoder** – bytes-to-bytes function, used on data chunks when building
- **encoderunit** – integer, encoder takes chunks of this size
- **sizecomputer** – function that computes amount of bytes outputed

Can propagate any exception from the lambda, possibly non-ConstructError. Can also raise arbitrary exceptions in RestreamedBytesIO implementation.

Example:

```
Bitwise  <--> Restreamed(subcon, bits2bytes, 8, bytes2bits, 1, lambda n: n//8)
Bytewise <--> Restreamed(subcon, bytes2bits, 1, bits2bytes, 8, lambda n: n*8)
```

construct.**ProcessXor**(*padfunc*, *subcon*)

Transforms bytes between the underlying stream and the subcon.

Used internally by KaitaiStruct compiler, when translating *process: xor* tags.

Parsing reads till EOF, xors data with the pad, then feeds that data into subcon. Building first builds the subcon into separate BytesIO stream, xors data with the pad, then writes that data into the main stream. Size is the same as subcon, unless it raises SizeofError.

Parameters

- **padfunc** – integer or bytes or context lambda, single or multiple bytes to xor data with
- **subcon** – Construct instance

Raises ***StringError*** – pad is not integer or bytes

Can propagate any exception from the lambda, possibly non-ConstructError.

Example:

```
>>> d = ProcessXor(0xf0 or b'\xf0', Int16ub)
>>> d.parse(b"\x00\xff")
0xf00f
>>> d.sizeof()
2
```

construct.**ProcessRotateLeft**(*amount*, *group*, *subcon*)

Transforms bytes between the underlying stream and the subcon.

Used internally by KaitaiStruct compiler, when translating *process: rol/ror* tags.

Parsing reads till EOF, rotates (shifts) the data *left* by amount in bits, then feeds that data into subcon. Building first builds the subcon into separate BytesIO stream, rotates *right* by negating amount, then writes that data into the main stream. Size is the same as subcon, unless it raises SizeofError.

Parameters

- **amount** – integer or context lambda, shift by this amount in bits, treated modulo (group x 8)
- **group** – integer or context lambda, shifting is applied to chunks of this size in bytes
- **subcon** – Construct instance

Raises

- ***RotationError*** – group is less than 1
- ***RotationError*** – data length is not a multiple of group size

Can propagate any exception from the lambda, possibly non-ConstructError.

Example:

```
>>> d = ProcessRotateLeft(4, 1, Int16ub)
>>> d.parse(b'\x0f\xf0')
0xf00f
>>> d = ProcessRotateLeft(4, 2, Int16ub)
>>> d.parse(b'\x0f\xf0')
```

(continues on next page)

(continued from previous page)

```
0xff00
>>> d.sizeof()
2
```

construct.**Checksum**(*checksumfield*, *hashfunc*, *bytesfunc*)

Field that is build or validated by a hash of a given byte range. Usually used with *RawCopy* .

Parsing compares parsed subcon *checksumfield* with a context entry provided by *bytesfunc* and transformed by *hashfunc*. Building fetches the contect entry, transforms it, then writes is using subcon. Size is same as subcon.

Parameters

- **checksumfield** – a subcon field that reads the checksum, usually Bytes(int)
- **hashfunc** – function that takes bytes and returns whatever checksumfield takes when building, usually from hashlib module
- **bytesfunc** – context lambda that returns bytes (or object) to be hashed, usually like this.rawcopy1.data

Raises ***ChecksumError*** – parsing and actual checksum does not match actual data

Can propagate any exception from the lambdas, possibly non-ConstructError.

Example:

```
import hashlib
d = Struct(
    "fields" / RawCopy(Struct(
        Padding(1000),
    )),
    "checksum" / Checksum(Bytes(64),
        lambda data: hashlib.sha512(data).digest(),
        this.fields.data),
)
d.build(dict(fields=dict(value={})))
```

```
import hashlib
d = Struct(
    "offset" / Tell,
    "checksum" / Padding(64),
    "fields" / RawCopy(Struct(
        Padding(1000),
    )),
    "checksum" / Pointer(this.offset, Checksum(Bytes(64),
        lambda data: hashlib.sha512(data).digest(),
        this.fields.data)),
)
d.build(dict(fields=dict(value={})))
```

construct.**Compressed**(*subcon*, *encoding*, *level=None*)

Compresses and decompresses underlying stream when processing subcon. When parsing, entire stream is consumed. When building, puts compressed bytes without marking the end. This construct should be used with *Prefixed*.

Parsing and building transforms all bytes using a specified codec. Since data is processed until EOF, it behaves similar to *GreedyBytes*. Size is undefined.

Parameters

- **subcon** – Construct instance, subcon used for storing the value
- **encoding** – string, any of module names like zlib/gzip/bzip2/lzma, otherwise any of codecs module bytes<->bytes encodings, each codec usually requires some Python version
- **level** – optional, integer between 0..9, although lzma discards it, some encoders allow different compression levels

Raises

- **ImportError** – needed module could not be imported by ctor
- ***StreamError*** – stream failed when reading until EOF

Example:

```
>>> d = Prefixed(VarInt, Compressed(GreedyBytes, "zlib"))
>>> d.build(bytes(100))
b'\x0cx\x9cc`\xa0=\x00\x00\x00d\x00\x01'
```

construct.**Rebuffered**(*subcon*, *tailcutoff=None*)

Caches bytes from underlying stream, so it becomes seekable and tellable, and also becomes blocking on reading. Useful for processing non-file streams like pipes, sockets, etc.

> **Warning:** Experimental implementation. May not be mature enough.

Parameters

- **subcon** – Construct instance, subcon which will operate on the buffered stream
- **tailcutoff** – optional, integer, amount of bytes kept in buffer, by default buffers everything

Can also raise arbitrary exceptions in its implementation.

Example:

```
Rebuffered(..., tailcutoff=1024).parse_stream(nonseekable_stream)
```

2.15 Core API: Lazy equivalents

construct.**Lazy**(*subcon*)

Lazyfies a field.

This wrapper allows you to do lazy parsing of individual fields inside a normal Struct (without using LazyStruct which may not work in every scenario). It is also used by KaitaiStruct compiler to emit *instances* because those are not processed greedily, and they may refer to other not yet parsed fields. Those are 2 entirely different applications but semantics are the same.

Parsing saves the current stream offset and returns a lambda. If and when that lambda gets evaluated, it seeks the stream to then-current position, parses the subcon, and seeks the stream back to previous position. Building evaluates that lambda into an object (if needed), then defers to subcon. Size also defers to subcon.

Parameters **subcon** – Construct instance

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- ***StreamError*** – stream is not seekable and tellable

Example:

```
>>> d = Lazy(Byte)
>>> x = d.parse(b'\x00')
>>> x
<function construct.core.Lazy._parse.<locals>.execute>
>>> x()
0
>>> d.build(0)
b'\x00'
>>> d.build(x)
b'\x00'
>>> d.sizeof()
1
```

construct.**LazyStruct**(**subcons*, ***subconskw*)

Equivalent to *Struct*, but when this class is parsed, most fields are not parsed (they are skipped if their size can be measured by _actualsize or _sizeof method). See its docstring for details.

Fields are parsed depending on some factors:

- Some fields like Int* Float* Bytes(5) Array(5,Byte) Pointer are fixed-size and are therefore skipped. Stream is not read.
- Some fields like Bytes(this.field) are variable-size but their size is known during parsing when there is a corresponding context entry. Those fields are also skipped. Stream is not read.
- Some fields like Prefixed PrefixedArray PascalString are variable-size but their size can be computed by partially reading the stream. Only first few bytes are read (the lengthfield).
- Other fields like VarInt need to be parsed. Stream position that is left after the field was parsed is used.
- Some fields may not work properly, due to the fact that this class attempts to skip fields, and parses them only out of necessity. Miscellaneous fields often have size defined as 0, and fixed sized fields are skippable.

Note there are restrictions:

- If a field like Bytes(this.field) references another field in the same struct, you need to access the referenced field first (to trigger its parsing) and then you can access the Bytes field. Otherwise it would fail due to missing context entry.
- If a field references another field within inner (nested) or outer (super) struct, things may break. Context is nested, but this class was not rigorously tested in that manner.

Building and sizeof are greedy, like in Struct.

Parameters

- ***subcons** – Construct instances, list of members, some can be anonymous
- ****subconskw** – Construct instances, list of members (requires Python 3.6)

construct.**LazyArray**(*count*, *subcon*)

Equivalent to *Array*, but the subcon is not parsed when possible (it gets skipped if the size can be measured by _actualsize or _sizeof method). See its docstring for details.

Fields are parsed depending on some factors:

- Some fields like Int* Float* Bytes(5) Array(5,Byte) Pointer are fixed-size and are therefore skipped. Stream is not read.
- Some fields like Bytes(this.field) are variable-size but their size is known during parsing when there is a corresponding context entry. Those fields are also skipped. Stream is not read.
- Some fields like Prefixed PrefixedArray PascalString are variable-size but their size can be computed by partially reading the stream. Only first few bytes are read (the lengthfield).
- Other fields like VarInt need to be parsed. Stream position that is left after the field was parsed is used.
- Some fields may not work properly, due to the fact that this class attempts to skip fields, and parses them only out of necessity. Miscellaneous fields often have size defined as 0, and fixed sized fields are skippable.

Note there are restrictions:

- If a field references another field within inner (nested) or outer (super) struct, things may break. Context is nested, but this class was not rigorously tested in that manner.

Building and sizeof are greedy, like in Array.

Parameters

- **count** – integer or context lambda, strict amount of elements
- **subcon** – Construct instance, subcon to process individual elements

construct.**LazyBound**(*subconfunc*)

Field that binds to the subcon only at runtime (during parsing and building, not ctor). Useful for recursive data structures, like linked-lists and trees, where a construct needs to refer to itself (while it does not exist yet in the namespace).

Note that it is possible to obtain same effect without using this class, using a loop. However there are usecases where that is not possible (if remaining nodes cannot be sized-up, and there is data following the recursive structure). There is also a significant difference, namely that LazyBound actually does greedy parsing while the loop does lazy parsing. See examples.

To break recursion, use *If* field. See examples.

Parameters **subconfunc** – parameter-less lambda returning Construct instance, can also return itself

Example:

```
d = Struct(
    "value" / Byte,
    "next" / If(this.value > 0, LazyBound(lambda: d)),
)
>>> print(d.parse(b"\x05\x09\x00"))
Container:
    value = 5
    next = Container:
        value = 9
        next = Container:
            value = 0
            next = None
```

```
d = Struct(
    "value" / Byte,
    "next" / GreedyBytes,
)
data = b"\x05\x09\x00"
```

(continues on next page)

(continued from previous page)

```
while data:
    x = d.parse(data)
    data = x.next
    print(x)
# print outputs
Container:
    value = 5
    next = \t\x00 (total 2)
# print outputs
Container:
    value = 9
    next = \x00 (total 1)
# print outputs
Container:
    value = 0
    next =  (total 0)
```

2.16 Core API: Debugging

construct.**Probe**(*into=None*, *lookahead=None*)

Probe that dumps the context, and some stream content (peeks into it) to the screen to aid the debugging process. It can optionally limit itself to a single context entry, instead of printing entire context.

Parameters

- **into** – optional, None by default, or context lambda
- **lookahead** – optional, integer, number of bytes to dump from the stream

Example:

```
>>> d = Struct(
...     "count" / Byte,
...     "items" / Byte[this.count],
...     Probe(lookahead=32),
... )
>>> d.parse(b"\x05abcde\x01\x02\x03")

--------------------------------------------------
Probe, path is (parsing), into is None
Stream peek: (hexlified) b'010203'...
Container:
    count = 5
    items = ListContainer:
        97
        98
        99
        100
        101
--------------------------------------------------
```

```
>>> d = Struct(
...     "count" / Byte,
...     "items" / Byte[this.count],
...     Probe(this.count),
```

(continues on next page)

(continued from previous page)

```
... )
>>> d.parse(b"\x05abcde\x01\x02\x03")

--------------------------------------------------
Probe, path is (parsing), into is this.count
5
--------------------------------------------------
```

construct.**setGlobalPrintFullStrings**(*enabled=False*)
When enabled, Container __str__ produces full content of bytes and unicode strings, otherwise and by default, it produces truncated output (16 bytes and 32 characters).

Parameters **enabled** – bool

construct.**setGlobalPrintFalseFlags**(*enabled=False*)
When enabled, Container __str__ that was produced by FlagsEnum parsing prints all values, otherwise and by default, it prints only the values that are True.

Parameters **enabled** – bool

construct.**setGlobalPrintPrivateEntries**(*enabled=False*)
When enabled, Container __str__ shows keys like _ _index _etc, otherwise and by default, it hides those keys. __repr__ never shows private entries.

Parameters **enabled** – bool

construct.**Debugger**(*subcon*)
PDB-based debugger. When an exception occurs in the subcon, a debugger will appear and allow you to debug the error (and even fix it on-the-fly).

Parameters **subcon** – Construct instance, subcon to debug

Example:

```
>>> Debugger(Byte[3]).build([])

--------------------------------------------------
Debugging exception of <Array: None>
path is (building)
  File "/media/arkadiusz/MAIN/GitHub/construct/construct/debug.py", line 192, in _
↪build
    return self.subcon._build(obj, stream, context, path)
  File "/media/arkadiusz/MAIN/GitHub/construct/construct/core.py", line 2149, in _
↪build
    raise RangeError("expected %d elements, found %d" % (count, len(obj)))
construct.core.RangeError: expected 3 elements, found 0

> /media/arkadiusz/MAIN/GitHub/construct/construct/core.py(2149)_build()
-> raise RangeError("expected %d elements, found %d" % (count, len(obj)))
(Pdb) q
--------------------------------------------------
```

2.17 Core API: Adapters and Validators

construct.**ExprAdapter**(*subcon*, *decoder*, *encoder*)
Generic adapter that takes *decoder* and *encoder* lambdas as parameters. You can use ExprAdapter instead of writing a full-blown class deriving from Adapter when only a simple lambda is needed.

Parameters

- **subcon** – Construct instance, subcon to adapt
- **decoder** – lambda that takes (obj, context) and returns an decoded version of obj
- **encoder** – lambda that takes (obj, context) and returns an encoded version of obj

Example:

```
>>> d = ExprAdapter(Byte, obj_+1, obj_-1)
>>> d.parse(b'\x04')
5
>>> d.build(5)
b'\x04'
```

construct.**ExprSymmetricAdapter**(*subcon*, *encoder*)

Macro around *ExprAdapter*.

Parameters

- **subcon** – Construct instance, subcon to adapt
- **encoder** – lambda that takes (obj, context) and returns both encoded version and decoded version of obj

Example:

```
>>> d = ExprSymmetricAdapter(Byte, obj_ & 0b00001111)
>>> d.parse(b"ÿ")
15
>>> d.build(255)
b''
```

construct.**ExprValidator**(*subcon*, *validator*)

Generic adapter that takes *validator* lambda as parameter. You can use ExprValidator instead of writing a full-blown class deriving from Validator when only a simple lambda is needed.

Parameters

- **subcon** – Construct instance, subcon to adapt
- **validator** – lambda that takes (obj, context) and returns a bool

Example:

```
>>> d = ExprValidator(Byte, obj_ & 0b11111110 == 0)
>>> d.build(1)
b'\x01'
>>> d.build(88)
ValidationError: object failed validation: 88
```

construct.**OneOf**(*subcon*, *valids*)

Validates that the object is one of the listed values, both during parsing and building.

Note: For performance, *valids* should be a set/frozenset.

Parameters

- **subcon** – Construct instance, subcon to validate

- **valids** – collection implementing __contains__, usually a list or set

Raises ***ValidationError*** – parsed or build value is not among valids

Example:

```
>>> d = OneOf(Byte, [1,2,3])
>>> d.parse(b"\x01")
1
>>> d.parse(b"\xff")
construct.core.ValidationError: object failed validation: 255
```

construct.**NoneOf**(*subcon*, *invalids*)

Validates that the object is none of the listed values, both during parsing and building.

Note: For performance, *valids* should be a set/frozenset.

Parameters

- **subcon** – Construct instance, subcon to validate
- **invalids** – collection implementing __contains__, usually a list or set

Raises ***ValidationError*** – parsed or build value is among invalids

construct.**Filter**(*predicate*, *subcon*)

Filters a list leaving only the elements that passed through the predicate.

Parameters

- **subcon** – Construct instance, usually Array GreedyRange Sequence
- **predicate** – lambda that takes (obj, context) and returns a bool

Can propagate any exception from the lambda, possibly non-ConstructError.

Example:

```
>>> d = Filter(obj_ != 0, Byte[:])
>>> d.parse(b"\x00\x02\x00")
[2]
>>> d.build([0,1,0,2,0])
b'\x01\x02'
```

construct.**Slicing**(*subcon*, *count*, *start*, *stop*, *step=1*, *empty=None*)

Adapter for slicing a list. Works with GreedyRange and Sequence.

Parameters

- **subcon** – Construct instance, subcon to slice
- **count** – integer, expected number of elements, needed during building
- **start** – integer for start index (or None for entire list)
- **stop** – integer for stop index (or None for up-to-end)
- **step** – integer, step (or 1 for every element)
- **empty** – object, value to fill the list with, during building

Example:

```
d = Slicing(Array(4,Byte), 4, 1, 3, empty=0)
assert d.parse(b"\x01\x02\x03\x04") == [2,3]
assert d.build([2,3]) == b"\x00\x02\x03\x00"
assert d.sizeof() == 4
```

construct.**Indexing**(*subcon*, *count*, *index*, *empty=None*)

Adapter for indexing a list (getting a single item from that list). Works with Range and Sequence and their lazy equivalents.

Parameters

- **subcon** – Construct instance, subcon to index
- **count** – integer, expected number of elements, needed during building
- **index** – integer, index of the list to get
- **empty** – object, value to fill the list with, during building

Example:

```
d = Indexing(Array(4,Byte), 4, 2, empty=0)
assert d.parse(b"\x01\x02\x03\x04") == 3
assert d.build(3) == b"\x00\x00\x03\x00"
assert d.sizeof() == 4
```

2.18 construct.core – entire module

exception construct.core.**AdaptationError**

class construct.core.**Adapter**(*subcon*)

Abstract adapter class.

Needs to implement *_decode()* for parsing and *_encode()* for building.

Parameters **subcon** – Construct instance

class construct.core.**Aligned**(*modulus*, *subcon*, *pattern='x00'*)

Appends additional null bytes to achieve a length that is shortest multiple of a modulus.

Note that subcon can actually be variable size, it is the eventual amount of bytes that is read or written during parsing or building that determines actual padding.

Parsing first parses subcon, then consumes an amount of bytes to sum up to specified length, and discards it. Building first builds subcon, then writes specified pattern byte to sum up to specified length. Size is subcon size plus modulo remainder, unless SizeofError was raised.

Parameters

- **modulus** – integer or context lambda, modulus to final length
- **subcon** – Construct instance
- **pattern** – optional, b-character, padding pattern, default is \x00

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- ***PaddingError*** – modulus was less than 2

- ***PaddingError*** – pattern was not bytes (b-character)

Can propagate any exception from the lambda, possibly non-ConstructError.

Example:

```
>>> d = Aligned(4, Int16ub)
>>> d.parse(b'\x00\x01\x00\x00')
1
>>> d.sizeof()
4
```

construct.core.**AlignedStruct**(*modulus*, **subcons*, ***subconskw*)

Makes a structure where each field is aligned to the same modulus (it is a struct of aligned fields, NOT an aligned struct).

See *Aligned* and *Struct* for semantics and raisable exceptions.

Parameters

- **modulus** – integer or context lambda, passed to each member
- ***subcons** – Construct instances, list of members, some can be anonymous
- ****subconskw** – Construct instances, list of members (requires Python 3.6)

Example:

```
>>> d = AlignedStruct(4, "a"/Int8ub, "b"/Int16ub)
>>> d.build(dict(a=0xFF,b=0xFFFF))
b'\xff\x00\x00\x00\xff\xff\x00\x00'
```

class construct.core.**Array**(*count*, *subcon*, *discard=False*)

Homogenous array of elements, similar to C# generic T[].

Parses into a ListContainer (a list). Parsing and building processes an exact amount of elements. If given list has more or less than count elements, raises RangeError. Size is defined as count multiplied by subcon size, but only if subcon is fixed size.

Operator [] can be used to make Array instances (recommended syntax).

Parameters

- **count** – integer or context lambda, strict amount of elements
- **subcon** – Construct instance, subcon to process individual elements
- **discard** – optional, bool, if set then parsing returns empty list

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- ***RangeError*** – specified count is not valid
- ***RangeError*** – given object has different length than specified count

Can propagate any exception from the lambdas, possibly non-ConstructError.

Example:

```
>>> d = Array(5, Byte) or Byte[5]
>>> d.build(range(5))
b'\x00\x01\x02\x03\x04'
>>> d.parse(_)
[0, 1, 2, 3, 4]
```

construct.core.**BitStruct**(**subcons*, ***subconskw*)

Makes a structure inside a Bitwise.

See *Bitwise* and *Struct* for semantics and raisable exceptions.

Parameters

- ***subcons** – Construct instances, list of members, some can be anonymous
- ****subconskw** – Construct instances, list of members (requires Python 3.6)

Example:

```
BitStruct  <-->  Bitwise(Struct(...))

>>> d = BitStruct(
...     "a" / Flag,
...     "b" / Nibble,
...     "c" / BitsInteger(10),
...     "d" / Padding(1),
... )
>>> d.parse(b"\xbe\xef")
Container(a=True)(b=7)(c=887)(d=None)
>>> d.sizeof()
2
```

class construct.core.**BitsInteger**(*length*, *signed=False*, *swapped=False*)

Field that packs arbitrarily large (or small) integers. Some fields (Bit Nibble Octet) use this class. Must be enclosed in *Bitwise* context.

Parses into an integer. Builds from an integer into specified bit count and endianness. Size (in bits) is specified in ctor.

Note that little-endianness is only defined for multiples of 8 bits.

Analog to *BytesInteger* that operates on bytes. In fact, `BytesInteger(n)` is equivalent to `Bitwise(BitsInteger(8*n))` and `BitsInteger(n)` is equivalent to `Bytewise(BytesInteger(n//8))`.

Parameters

- **length** – integer or context lambda, number of bits in the field
- **signed** – bool, whether the value is signed (two's complement), default is False (unsigned)
- **swapped** – bool, whether to swap byte order (little endian), default is False (big endian)

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- ***IntegerError*** – lenght is negative, given a negative value when field is not signed, or not an integer

Can propagate any exception from the lambda, possibly non-ConstructError.

Example:

```
>>> d = Bitwise(BitsInteger(8)) or Bitwise(Octet)
>>> d.parse(b"\x10")
16
>>> d.build(255)
b'\xff'
>>> d.sizeof()
1
```

construct.core.**BitsSwapped**(*subcon*)

Swaps the bit order within each byte within boundaries of given subcon. Does NOT require a fixed sized subcon.

Parameters **subcon** – Construct instance, subcon on top of bit swapped bytes

Raises ***SizeofError*** – compiler could not compute subcon size

See *Transformed* and *Restreamed* for raisable exceptions.

Example:

```
>>> d = Bitwise(Bytes(8))
>>> d.parse(b"\x01")
'\x00\x00\x00\x00\x00\x00\x00\x01'
>>>> BitsSwapped(d).parse(b"\x01")
'\x01\x00\x00\x00\x00\x00\x00\x00'
```

class construct.core.**BitwisableString**

Used internally.

construct.core.**Bitwise**(*subcon*)

Converts the stream from bytes to bits, and passes the bitstream to underlying subcon. Bitstream is a stream that contains 8 times as many bytes, and each byte is either \x00 or \x01 (in documentation those bytes are called bits).

Parsing building and size are deferred to subcon, although size gets divided by 8.

Parameters **subcon** – Construct instance, any field that works with bits (like BitsInteger) or is bit-byte agnostic (like Struct or Flag)

See *Transformed* and *Restreamed* for raisable exceptions.

Example:

```
>>> d = Bitwise(Struct(
...     'a' / Nibble,
...     'b' / Bytewise(Float32b),
...     'c' / Padding(4),
... ))
>>> d.parse(bytes(5))
Container(a=0)(b=0.0)(c=None)
>>> d.sizeof()
5
```

construct.core.**ByteSwapped**(*subcon*)

Swaps the byte order within boundaries of given subcon. Requires a fixed sized subcon.

Parameters **subcon** – Construct instance, subcon on top of byte swapped bytes

Raises ***SizeofError*** – ctor or compiler could not compute subcon size

See *Transformed* and *Restreamed* for raisable exceptions.

Example:

```
Int24ul <--> ByteSwapped(Int24ub) <--> BytesInteger(3, swapped=True) <-->␣
↪ByteSwapped(BytesInteger(3))
```

class construct.core.**Bytes**(*length*)

Field consisting of a specified number of bytes.

Parses into a bytes (of given length). Builds into the stream directly (but checks that given object matches specified length). Can also build from an integer for convenience (although BytesInteger should be used instead). Size is the specified length.

Parameters **length** – integer or context lambda

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- ***StringError*** – building from non-bytes value, perhaps unicode

Can propagate any exception from the lambda, possibly non-ConstructError.

Example:

```
>>> d = Bytes(4)
>>> d.parse(b'beef')
b'beef'
>>> d.build(b'beef')
b'beef'
>>> d.build(0)
b'\x00\x00\x00\x00'
>>> d.sizeof()
4

>>> d = Struct(
...     "length" / Int8ub,
...     "data" / Bytes(this.length),
... )
>>> d.parse(b"\x04beef")
Container(length=4)(data=b'beef')
>>> d.sizeof()
construct.core.SizeofError: cannot calculate size, key not found in context
```

class construct.core.**BytesInteger**(*length*, *signed=False*, *swapped=False*)

Field that packs arbitrarily large integers. Some Int24* fields use this class.

Parses into an integer. Builds from an integer into specified byte count and endianness. Size is specified in ctor.

Analog to *BitsInteger* that operates on bits. In fact, `BytesInteger(n)` is equivalent to `Bitwise(BitsInteger(8*n))` and `BitsInteger(n)` is equivalent to `Bytewise(BytesInteger(n//8))`.

Parameters

- **length** – integer or context lambda, number of bytes in the field
- **signed** – bool, whether the value is signed (two's complement), default is False (unsigned)
- **swapped** – bool, whether to swap byte order (little endian), default is False (big endian)

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- ***IntegerError*** – lenght is negative, given a negative value when field is not signed, or not an integer

Can propagate any exception from the lambda, possibly non-ConstructError.

Example:

```
>>> d = BytesInteger(4) or Int32ub
>>> d.parse(b"abcd")
1633837924
>>> d.build(1)
b'\x00\x00\x00\x01'
>>> d.sizeof()
4
```

construct.core.**Bytewise**(*subcon*)

Converts the bitstream back to normal byte stream. Must be used within *Bitwise*.

Parsing building and size are deferred to subcon, although size gets multiplied by 8.

Parameters **subcon** – Construct instance, any field that works with bytes or is bit-byte agnostic

See *Transformed* and *Restreamed* for raisable exceptions.

Example:

```
>>> d = Bitwise(Struct(
...     'a' / Nibble,
...     'b' / Bytewise(Float32b),
...     'c' / Padding(4),
... ))
>>> d.parse(bytes(5))
Container(a=0)(b=0.0)(c=None)
>>> d.sizeof()
5
```

construct.core.**CString**(*encoding*)

String ending in a terminating null byte (or null bytes in case of UTF16 UTF32).

Warning: String and CString only support encodings explicitly listed in *possiblestringencodings* .

Parameters **encoding** – string like: utf8 utf16 utf32 ascii

Raises

- ***StringError*** – building a non-unicode string
- ***StringError*** – selected encoding is not on supported list

Example:

```
>>> d = CString("utf8")
>>> d.build(u"")
b'\xd0\x90\xd1\x84\xd0\xbe\xd0\xbd\x00'
>>> d.parse(_)
u''
```

exception construct.core.**CancelParsing**

class construct.core.**Check**(*func*)

Checks for a condition, and raises CheckError if the check fails.

Parsing and building return nothing (but check the condition). Size is 0 because stream is unaffected.

Parameters **func** – bool or context lambda, that gets run on parsing and building

Raises ***CheckError*** – lambda returned false

Can propagate any exception from the lambda, possibly non-ConstructError.

Example:

```
Check(lambda ctx: len(ctx.payload.data) == ctx.payload_len)
Check(len_(this.payload.data) == this.payload_len)
```

exception construct.core.**CheckError**

class construct.core.**Checksum**(*checksumfield*, *hashfunc*, *bytesfunc*)

Field that is build or validated by a hash of a given byte range. Usually used with *RawCopy* .

Parsing compares parsed subcon *checksumfield* with a context entry provided by *bytesfunc* and transformed by *hashfunc*. Building fetches the contect entry, transforms it, then writes is using subcon. Size is same as subcon.

Parameters

- **checksumfield** – a subcon field that reads the checksum, usually Bytes(int)
- **hashfunc** – function that takes bytes and returns whatever checksumfield takes when building, usually from hashlib module
- **bytesfunc** – context lambda that returns bytes (or object) to be hashed, usually like this.rawcopy1.data

Raises ***ChecksumError*** – parsing and actual checksum does not match actual data

Can propagate any exception from the lambdas, possibly non-ConstructError.

Example:

```
import hashlib
d = Struct(
    "fields" / RawCopy(Struct(
        Padding(1000),
    )),
    "checksum" / Checksum(Bytes(64),
        lambda data: hashlib.sha512(data).digest(),
        this.fields.data),
)
d.build(dict(fields=dict(value={})))
```

```
import hashlib
d = Struct(
    "offset" / Tell,
    "checksum" / Padding(64),
    "fields" / RawCopy(Struct(
        Padding(1000),
    )),
    "checksum" / Pointer(this.offset, Checksum(Bytes(64),
        lambda data: hashlib.sha512(data).digest(),
```

(continues on next page)

(continued from previous page)

```
        this.fields.data)),
)
d.build(dict(fields=dict(value={})))
```

exception construct.core.**ChecksumError**

class construct.core.**Compiled**(*source*, *defersubcon*, *parsefunc*)

Used internally.

benchmark(*sampledata*, *filename=None*)

Measures performance of your construct (its parsing and building runtime), both for the original instance and the compiled instance. Uses timeit module, over at min 1 sample, and at max over 1 second time.

Optionally, results are saved to a text file for later inspection. Otherwise you can print the result string to terminal.

Also this method checks correctness, by comparing parsing/building results from both instances.

Parameters

- **sampledata** – bytes, a valid blob parsable by this construct
- **filename** – optional, string, source is saved to that file

Returns string containing measurements

compile(*filename=None*)

Transforms a construct into another construct that does same thing (has same parsing and building semantics) but is much faster when parsing. Already compiled instances just compile into itself.

Optionally, partial source code can be saved to a text file. This is meant only to inspect the generated code, not to import it from external scripts.

Returns Compiled instance

class construct.core.**Compressed**(*subcon*, *encoding*, *level=None*)

Compresses and decompresses underlying stream when processing subcon. When parsing, entire stream is consumed. When building, puts compressed bytes without marking the end. This construct should be used with *Prefixed*.

Parsing and building transforms all bytes using a specified codec. Since data is processed until EOF, it behaves similar to *GreedyBytes*. Size is undefined.

Parameters

- **subcon** – Construct instance, subcon used for storing the value
- **encoding** – string, any of module names like zlib/gzip/bzip2/lzma, otherwise any of codecs module bytes<->bytes encodings, each codec usually requires some Python version
- **level** – optional, integer between 0..9, although lzma discards it, some encoders allow different compression levels

Raises

- **ImportError** – needed module could not be imported by ctor
- ***StreamError*** – stream failed when reading until EOF

Example:

```
>>> d = Prefixed(VarInt, Compressed(GreedyBytes, "zlib"))
>>> d.build(bytes(100))
b'\x0cx\x9cc`\xa0=\x00\x00\x00d\x00\x01'
```

class construct.core.**Computed**(*func*)

Field computing a value from the context dictionary or some outer source like os.urandom or random module. Underlying byte stream is unaffected. The source can be non-deterministic.

Parsing and Building return the value returned by the context lambda (although a constant value can also be used). Size is defined as 0 because parsing and building does not consume or produce bytes into the stream.

Parameters **func** – context lambda or constant value

Can propagate any exception from the lambda, possibly non-ConstructError.

Example::

```
>>> d = Struct(
...     "width" / Byte,
...     "height" / Byte,
...     "total" / Computed(this.width * this.height),
... )
>>> d.build(dict(width=4,height=5))
b'\x04\x05'
>>> d.parse(b"12")
Container(width=49, height=50, total=2450)
```

```
>>> d = Computed(7)
>>> d.parse(b"")
7
>>> d = Computed(lambda ctx: 7)
>>> d.parse(b"")
7
```

```
>>> import os
>>> d = Computed(lambda ctx: os.urandom(10))
>>> d.parse(b"")
b'\x98\xc2\xec\x10\x07\xf5\x8e\x98\xc2\xec'
```

class construct.core.**Const**(*value*, *subcon=None*)

Field enforcing a constant. It is used for file signatures, to validate that the given pattern exists. Data in the stream must strictly match the specified value.

Note that a variable sized subcon may still provide positive verification. Const does not consume a precomputed amount of bytes, but depends on the subcon to read the appropriate amount (eg. VarInt is acceptable). Whatever subcon parses into, gets compared against the specified value.

Parses using subcon and return its value (after checking). Builds using subcon from nothing (or given object, if not None). Size is the same as subcon, unless it raises SizeofError.

Parameters

- **value** – expected value, usually a bytes literal
- **subcon** – optional, Construct instance, subcon used to build value from, assumed to be Bytes if value parameter was a bytes literal

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- ***ConstError*** – parsed data does not match specified value, or building from wrong value
- ***StringError*** – building from non-bytes value, perhaps unicode

Example:

```
>>> d = Const(b"IHDR")
>>> d.build(None)
b'IHDR'
>>> d.parse(b"JPEG")
construct.core.ConstError: expected b'IHDR' but parsed b'JPEG'

>>> d = Const(255, Int32ul)
>>> d.build(None)
b'\xff\x00\x00\x00'
```

exception construct.core.**ConstError**

class construct.core.**Construct**

The mother of all constructs.

This object is generally not directly instantiated, and it does not directly implement parsing and building, so it is largely only of interest to subclass implementors. There are also other abstract classes sitting on top of this one.

The external user API:

- *parse*
- *parse_stream*
- *parse_file*
- *build*
- *build_stream*
- *build_file*
- *sizeof*
- *compile*
- *benchmark*

Subclass authors should not override the external methods. Instead, another API is available:

- *_parse*
- *_build*
- *_sizeof*
- *_actualsize*
- *_emitparse*
- *_emitbuild*
- *__getstate__*
- *__setstate__*

Attributes and Inheritance:

All constructs have a name and flags. The name is used for naming struct members and context dictionaries. Note that the name can be a string, or None by default. A single underscore "_" is a reserved name, used as up-level in nested containers. The name should be descriptive, short, and valid as a Python identifier, although these rules are not enforced. The flags specify additional behavioral information about this construct. Flags are used by enclosing constructs to determine a proper course of action. Flags are often inherited from inner subconstructs but that depends on each class.

benchmark(*sampledata*, *filename=None*)

Measures performance of your construct (its parsing and building runtime), both for the original instance and the compiled instance. Uses timeit module, over at min 1 sample, and at max over 1 second time.

Optionally, results are saved to a text file for later inspection. Otherwise you can print the result string to terminal.

Also this method checks correctness, by comparing parsing/building results from both instances.

Parameters

- **sampledata** – bytes, a valid blob parsable by this construct
- **filename** – optional, string, source is saved to that file

Returns string containing measurements

build(*obj*, ***contextkw*)

Build an object in memory (a bytes object).

Whenever data cannot be written, ConstructError or its derivative is raised. This method is NOT ALLOWED to raise any other exceptions although (1) user-defined lambdas can raise arbitrary exceptions which are propagated (2) external libraries like numpy can raise arbitrary exceptions which are propagated (3) some list and dict lookups can raise IndexError and KeyError which are propagated.

Context entries are passed only as keyword parameters **contextkw.

Parameters ****contextkw** – context entries, usually empty

Returns bytes

Raises ***ConstructError*** – raised for any reason

build_file(*obj*, *filename*, ***contextkw*)

Build an object into a closed binary file. See build().

build_stream(*obj*, *stream*, ***contextkw*)

Build an object directly into a stream. See build().

compile(*filename=None*)

Transforms a construct into another construct that does same thing (has same parsing and building semantics) but is much faster when parsing. Already compiled instances just compile into itself.

Optionally, partial source code can be saved to a text file. This is meant only to inspect the generated code, not to import it from external scripts.

Returns Compiled instance

parse(*data*, ***contextkw*)

Parse an in-memory buffer (often bytes object). Strings, buffers, memoryviews, and other complete buffers can be parsed with this method.

Whenever data cannot be read, ConstructError or its derivative is raised. This method is NOT ALLOWED to raise any other exceptions although (1) user-defined lambdas can raise arbitrary exceptions which are

propagated (2) external libraries like numpy can raise arbitrary exceptions which are propagated (3) some list and dict lookups can raise IndexError and KeyError which are propagated.

Context entries are passed only as keyword parameters **contextkw.

Parameters ****contextkw** – context entries, usually empty

Returns some value, usually based on bytes read from the stream but sometimes it is computed from nothing or from the context dictionary, sometimes its non-deterministic

Raises ***ConstructError*** – raised for any reason

parse_file(*filename*, ***contextkw*)
Parse a closed binary file. See parse().

parse_stream(*stream*, ***contextkw*)
Parse a stream. Files, pipes, sockets, and other streaming sources of data are handled by this method. See parse().

sizeof(***contextkw*)
Calculate the size of this object, optionally using a context.

Some constructs have fixed size (like FormatField), some have variable-size and can determine their size given a context entry (like Bytes(this.otherfield1)), and some cannot determine their size (like VarInt).

Whenever size cannot be determined, SizeofError is raised. This method is NOT ALLOWED to raise any other exception, even if eg. context dictionary is missing a key, or subcon propagates ConstructError-derivative exception.

Context entries are passed only as keyword parameters **contextkw.

Parameters ****contextkw** – context entries, usually empty

Returns integer if computable, SizeofError otherwise

Raises ***SizeofError*** – size could not be determined in actual context, or is impossible to be determined

exception construct.core.**ConstructError**

class construct.core.**Default**(*subcon*, *value*)
Field where building does not require a value, because the value gets taken from default. Comes handy when building a Struct from a dict with missing keys.

Parsing defers to subcon. Building is defered to subcon, but it builds from a default (if given object is None) or from given object. Building does not require a value, but can accept one. Size is the same as subcon, unless it raises SizeofError.

Difference between Default and Rebuild, is that in first the build value is optional and in second the build value is ignored.

Parameters

- **subcon** – Construct instance
- **value** – context lambda or constant value

Raises ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes

Can propagate any exception from the lambda, possibly non-ConstructError.

Example:

```
>>> d = Struct(
...     "a" / Default(Byte, 0),
... )
>>> d.build(dict(a=1))
b'\x01'
>>> d.build(dict())
b'\x00'
```

class construct.core.**Embedded**(*subcon*)

Special wrapper that allows outer multiple-subcons construct to merge fields from another multiple-subcons construct. Embedded does not change a field, only wraps it like a candy with a flag.

Warning: Can only be used between Struct Sequence FocusedSeq Union LazyStruct, although they can be used interchangably, for example Struct can embed fields from a Sequence. There is also *EmbeddedSwitch* macro that pseudo-embeds a Switch. Its not possible to embed IfThenElse.

Parsing building and sizeof are deferred to subcon.

Parameters **subcon** – Construct instance, its fields to embed inside a struct or sequence

Example:

```
>>> outer = Struct(
...     Embedded(Struct(
...         "data" / Bytes(4),
...     )),
... )
>>> outer.parse(b"1234")
Container(data=b'1234')
```

construct.core.**EmbeddedSwitch**(*merged*, *selector*, *mapping*)

Macro that simulates embedding Switch, which under new embedding semantics is not possible. This macro does NOT produce a Switch. It generates classes that behave the same way as you would expect from embedded Switch, only that. Instance created by this macro CAN be embedded.

Both *merged* and all values in *mapping* must be Struct instances. Macro re-creates a single struct that contains all fields, where each field is wrapped in *If(selector == key, ...)*. Note that resulting dictionary contains None values for fields that would not be chosen by switch. Note also that if selector does not match any cases, it passes successfully (default Switch behavior).

All fields should have unique names. Otherwise fields that were not selected during parsing may return None and override other fields context entries that have same name. This is because *If* field returns None value if condition is not met, but the Struct inserts that None value into the context entry regardless.

Parameters

- **merged** – Struct instance
- **selector** – this expression, that references one of *merged* fields
- **mapping** – dict with values being Struct instances

Example:

```
d = EmbeddedSwitch(
    Struct(
        "type" / Byte,
    ),
```

(continues on next page)

(continued from previous page)

```
    this.type,
    {
        0: Struct("name" / PascalString(Byte, "utf8")),
        1: Struct("value" / Byte),
    }
)

# generates essentially following
d = Struct(
    "type" / Byte,
    "name" / If(this.type == 0, PascalString(Byte, "utf8")),
    "value" / If(this.type == 1, Byte),
)

# both parse like following
>>> d.parse(b"\x00\x00")
Container(type=0, name=u'', value=None)
>>> d.parse(b"\x01\x00")
Container(type=1, name=None, value=0)
```

class construct.core.**Enum**(*subcon*, **merge*, ***mapping*)

Translates unicode label names to subcon values, and vice versa.

Parses integer subcon, then uses that value to lookup mapping dictionary. Returns an integer-convertible string (if mapping found) or an integer (otherwise). Building is a reversed process. Can build from an integer flag or string label. Size is same as subcon, unless it raises SizeofError.

There is no default parameter, because if no mapping is found, it parses into an integer without error.

This class supports enum34 module. See examples.

This class supports exposing member labels as attributes, as integer-convertible strings. See examples.

Parameters

- **subcon** – Construct instance, subcon to map to/from
- ***merge** – optional, list of enum.IntEnum and enum.IntFlag instances, to merge labels and values from
- ****mapping** – dict, mapping string names to values

Raises ***MappingError*** – building from string but no mapping found

Example:

```
>>> d = Enum(Byte, one=1, two=2, four=4, eight=8)
>>> d.parse(b"\x01")
'one'
>>> int(d.parse(b"\x01"))
1
>>> d.parse(b"\xff")
255
>>> int(d.parse(b"\xff"))
255

>>> d.build(d.one or "one" or 1)
b'\x01'
>>> d.one
'one'
```

(continues on next page)

(continued from previous page)

```
import enum
class E(enum.IntEnum or enum.IntFlag):
    one = 1
    two = 2

Enum(Byte, E) <--> Enum(Byte, one=1, two=2)
FlagsEnum(Byte, E) <--> FlagsEnum(Byte, one=1, two=2)
```

class construct.core.**EnumInteger**
Used internally.

class construct.core.**EnumIntegerString**
Used internally.

exception construct.core.**ExplicitError**

class construct.core.**ExprAdapter**(*subcon*, *decoder*, *encoder*)
Generic adapter that takes *decoder* and *encoder* lambdas as parameters. You can use ExprAdapter instead of writing a full-blown class deriving from Adapter when only a simple lambda is needed.

Parameters

- **subcon** – Construct instance, subcon to adapt
- **decoder** – lambda that takes (obj, context) and returns an decoded version of obj
- **encoder** – lambda that takes (obj, context) and returns an encoded version of obj

Example:

```
>>> d = ExprAdapter(Byte, obj_+1, obj_-1)
>>> d.parse(b'\x04')
5
>>> d.build(5)
b'\x04'
```

class construct.core.**ExprSymmetricAdapter**(*subcon*, *encoder*)
Macro around *ExprAdapter*.

Parameters

- **subcon** – Construct instance, subcon to adapt
- **encoder** – lambda that takes (obj, context) and returns both encoded version and decoded version of obj

Example:

```
>>> d = ExprSymmetricAdapter(Byte, obj_ & 0b00001111)
>>> d.parse(b"ÿ")
15
>>> d.build(255)
b''
```

class construct.core.**ExprValidator**(*subcon*, *validator*)
Generic adapter that takes *validator* lambda as parameter. You can use ExprValidator instead of writing a full-blown class deriving from Validator when only a simple lambda is needed.

Parameters

- **subcon** – Construct instance, subcon to adapt

- **validator** – lambda that takes (obj, context) and returns a bool

Example:

```
>>> d = ExprValidator(Byte, obj_ & 0b11111110 == 0)
>>> d.build(1)
b'\x01'
>>> d.build(88)
ValidationError: object failed validation: 88
```

construct.core.**Filter**(*predicate*, *subcon*)

Filters a list leaving only the elements that passed through the predicate.

Parameters

- **subcon** – Construct instance, usually Array GreedyRange Sequence
- **predicate** – lambda that takes (obj, context) and returns a bool

Can propagate any exception from the lambda, possibly non-ConstructError.

Example:

```
>>> d = Filter(obj_ != 0, Byte[:])
>>> d.parse(b"\x00\x02\x00")
[2]
>>> d.build([0,1,0,2,0])
b'\x01\x02'
```

class construct.core.**FixedSized**(*length*, *subcon*)

Restricts parsing to specified amount of bytes.

Parsing reads *length* bytes, then defers to subcon using new BytesIO with said bytes. Building builds the subcon using new BytesIO, then writes said data and additional null bytes accordingly. Size is same as *length*, although negative amount raises an error as well.

Parameters

- **length** – integer or context lambda, total amount of bytes (both data and padding)
- **subcon** – Construct instance

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- ***PaddingError*** – length is negative
- ***PaddingError*** – subcon written more bytes than entire length (negative padding)

Can propagate any exception from the lambda, possibly non-ConstructError.

Example:

```
>>> d = FixedSized(10, Byte)
>>> d.parse(b'\xff\x00\x00\x00\x00\x00\x00\x00\x00\x00')
255
>>> d.build(255)
b'\xff\x00\x00\x00\x00\x00\x00\x00\x00\x00'
>>> d.sizeof()
10
```

class construct.core.**FlagsEnum**(*subcon*, **merge*, ***flags*)

Translates unicode label names to subcon integer (sub)values, and vice versa.

Parses integer subcon, then creates a Container, where flags define each key. Builds from a container by bitwise-oring of each flag if it matches a set key. Can build from an integer flag or string label directly, as well as | concatenations thereof (see examples). Size is same as subcon, unless it raises SizeofError.

This class supports enum34 module. See examples.

This class supports exposing member labels as attributes, as bitwisable strings. See examples.

Parameters

- **subcon** – Construct instance, must operate on integers
- ***merge** – optional, list of enum.IntEnum and enum.IntFlag instances, to merge labels and values from
- ****flags** – dict, mapping string names to integer values

Raises

- ***MappingError*** – building from object not like: integer string dict
- ***MappingError*** – building from string but no mapping found

Can raise arbitrary exceptions when computing | and & and value is non-integer.

Example:

```
>>> d = FlagsEnum(Byte, one=1, two=2, four=4, eight=8)
>>> d.parse(b"\x03")
Container(one=True, two=True, four=False, eight=False)
>>> d.build(dict(one=True,two=True))
b'\x03'

>>> d.build(d.one|d.two or "one|two" or 1|2)
b'\x03'

import enum
class E(enum.IntEnum or enum.IntFlag):
    one = 1
    two = 2

Enum(Byte, E) <--> Enum(Byte, one=1, two=2)
FlagsEnum(Byte, E) <--> FlagsEnum(Byte, one=1, two=2)
```

class construct.core.**FocusedSeq**(*parsebuildfrom*, **subcons*, ***subconskw*)

Allows constructing more elaborate "adapters" than Adapter class.

Parse does parse all subcons in sequence, but returns only the element that was selected (discards other values). Build does build all subcons in sequence, where each gets build from nothing (except the selected subcon which is given the object). Size is the sum of all subcon sizes, unless any subcon raises SizeofError.

This class does context nesting, meaning its members are given access to a new dictionary where the "_" entry points to the outer context. When parsing, each member gets parsed and subcon parse return value is inserted into context under matching key only if the member was named. When building, the matching entry gets inserted into context before subcon gets build, and if subcon build returns a new value (not None) that gets replaced in the context.

This class supports embedding. *Embedded* semantics dictate, that during instance creation (in ctor), each field is checked for embedded flag, and its subcon members are merged. This changes behavior of some code

examples. Only few classes are supported: Struct Sequence FocusedSeq Union LazyStruct, although those can be used interchangably (a Struct can embed a Sequence, or rather its members).

This class exposes subcons as attributes. You can refer to subcons that were inlined (and therefore do not exist as variable in the namespace) by accessing the struct attributes, under same name. Also note that compiler does not support this feature. See examples.

This class exposes subcons in the context. You can refer to subcons that were inlined (and therefore do not exist as variable in the namespace) within other inlined fields using the context. Note that you need to use a lambda (*this* expression is not supported). Also note that compiler does not support this feature. See examples.

This class is used internally to implement *PrefixedArray*.

Parameters

- **parsebuildfrom** – string name or context lambda, selects a subcon
- ***subcons** – Construct instances, list of members, some can be named
- ****subconskw** – Construct instances, list of members (requires Python 3.6)

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- **UnboundLocalError** – selector does not match any subcon

Can propagate any exception from the lambda, possibly non-ConstructError.

Excample:

```
>>> d = FocusedSeq("num", Const(b"SIG"), "num"/Byte, Terminated)
>>> d.parse(b"SIG\xff")
255
>>> d.build(255)
b'SIG\xff'

>>> d = FocusedSeq("animal",
...     "animal" / Enum(Byte, giraffe=1),
... )
>>> d.animal.giraffe
'giraffe'
>>> d = FocusedSeq("count",
...     "count" / Byte,
...     "data" / Padding(lambda this: this.count - this._subcons.count.sizeof()),
... )
>>> d.build(4)
b'\x04\x00\x00\x00'

PrefixedArray <--> FocusedSeq("items",
    "count" / Rebuild(lengthfield, len_(this.items)),
    "items" / subcon[this.count],
)
```

class construct.core.**FormatField**(*endianity*, *format*)

Field that uses *struct* module to pack and unpack CPU-sized integers and floats. This is used to implement most Int* Float* fields, but for example cannot pack 24-bit integers, which is left to *BytesInteger* class.

See struct module documentation for instructions on crafting format strings.

Parses into an integer. Builds from an integer into specified byte count and endianness. Size is determined by *struct* module according to specified format string.

Parameters

- **endianity** – string, character like: < > =
- **format** – string, character like: f d B H L Q b h l q

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- ***FormatFieldError*** – wrong format string, or struct.(un)pack complained about the value

Example:

```
>>> d = FormatField(">", "H") or Int16ub
>>> d.parse(b"\x01\x00")
256
>>> d.build(256)
b"\x01\x00"
>>> d.sizeof()
2
```

exception construct.core.**FormatFieldError**

class construct.core.**GreedyRange**(*subcon*, *discard=False*)

Homogenous array of elements, similar to C# generic IEnumerable<T>, but works with unknown count of elements by parsing until end of stream.

Parses into a ListContainer (a list). Parsing stops when an exception occured when parsing the subcon, either due to EOF or subcon format not being able to parse the data. Either way, when GreedyRange encounters either failure it seeks the stream back to a position after last successful subcon parsing. Builds from enumerable, each element as-is. Size is undefined.

This class supports stopping. If *StopIf* field is a member, and it evaluates its lambda as positive, this class ends parsing or building as successful without processing further fields.

Parameters

- **subcon** – Construct instance, subcon to process individual elements
- **discard** – optional, bool, if set then parsing returns empty list

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- ***StreamError*** – stream is not seekable and tellable

Can propagate any exception from the lambdas, possibly non-ConstructError.

Example:

```
>>> d = GreedyRange(Byte)
>>> d.build(range(8))
b'\x00\x01\x02\x03\x04\x05\x06\x07'
>>> d.parse(_)
[0, 1, 2, 3, 4, 5, 6, 7]
```

construct.core.**GreedyString**(*encoding*)

String that reads entire stream until EOF, and writes a given string as-is. Analog to GreedyBytes but also applies unicode-to-bytes encoding.

Parameters **encoding** – string like: utf8 utf16 utf32 ascii

Raises

- ***StringError*** – building a non-unicode string
- ***StreamError*** – stream failed when reading until EOF

Example:

```
>>> d = GreedyString("utf8")
>>> d.build(u"")
b'\xd0\x90\xd1\x84\xd0\xbe\xd0\xbd'
>>> d.parse(_)
u''
```

class construct.core.**Hex**(*subcon*)

Adapter for displaying hexadecimal/hexlified representation of integers/bytes/RawCopy dictionaries.

Parsing results in int-alike bytes-alike or dict-alike object, whose only difference from original is pretty-printing. If you look at the result, you will be presented with its *repr* which remains as-is. If you print it, then you will see its *str* whic is a hexlified representation. Building and sizeof defer to subcon.

To obtain a hexlified string (like before Hex HexDump changed semantics) use binascii.(un)hexlify on parsed results.

Example:

```
>>> d = Hex(Int32ub)
>>> obj = d.parse(b"\x00\x00\x01\x02")
>>> obj
258
>>> print(obj)
0x00000102

>>> d = Hex(GreedyBytes)
>>> obj = d.parse(b"\x00\x00\x01\x02")
>>> obj
b'\x00\x00\x01\x02'
>>> print(obj)
unhexlify('00000102')

>>> d = Hex(RawCopy(Int32ub))
>>> obj = d.parse(b"\x00\x00\x01\x02")
>>> obj
{'data': b'\x00\x00\x01\x02',
 'length': 4,
 'offset1': 0,
 'offset2': 4,
 'value': 258}
>>> print(obj)
unhexlify('00000102')
```

class construct.core.**HexDump**(*subcon*)

Adapter for displaying hexlified representation of bytes/RawCopy dictionaries.

Parsing results in bytes-alike or dict-alike object, whose only difference from original is pretty-printing. If you look at the result, you will be presented with its *repr* which remains as-is. If you print it, then you will see its *str* whic is a hexlified representation. Building and sizeof defer to subcon.

To obtain a hexlified string (like before Hex HexDump changed semantics) use construct.lib.hexdump on parsed results.

Example:

```
>>> d = HexDump(GreedyBytes)
>>> obj = d.parse(b"\x00\x00\x01\x02")
>>> obj
b'\x00\x00\x01\x02'
>>> print(obj)
hexundump('''
0000   00 00 01 02                                       ....
''')

>>> d = HexDump(RawCopy(Int32ub))
>>> obj = d.parse(b"\x00\x00\x01\x02")
>>> obj
{'data': b'\x00\x00\x01\x02',
 'length': 4,
 'offset1': 0,
 'offset2': 4,
 'value': 258}
>>> print(obj)
hexundump('''
0000   00 00 01 02                                       ....
''')
```

construct.core.**If**(*condfunc*, *subcon*)

If-then conditional construct.

Parsing evaluates condition, if True then subcon is parsed, otherwise just returns None. Building also evaluates condition, if True then subcon gets build from, otherwise does nothing. Size is either same as subcon or 0, depending how condfunc evaluates.

Parameters

- **condfunc** – bool or context lambda (or a truthy value)
- **subcon** – Construct instance, used if condition indicates True

Raises ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes

Can propagate any exception from the lambda, possibly non-ConstructError.

Example:

```
If <--> IfThenElse(condfunc, subcon, Pass)

>>> d = If(this.x > 0, Byte)
>>> d.build(255, x=1)
b'\xff'
>>> d.build(255, x=0)
b''
```

class construct.core.**IfThenElse**(*condfunc*, *thensubcon*, *elsesubcon*)

If-then-else conditional construct, similar to ternary operator.

Parsing and building evaluates condition, and defers to either subcon depending on the value. Size is computed the same way.

Parameters

- **condfunc** – bool or context lambda (or a truthy value)
- **thensubcon** – Construct instance, used if condition indicates True
- **elsesubcon** – Construct instance, used if condition indicates False

Raises ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes

Can propagate any exception from the lambda, possibly non-ConstructError.

Example:

```
>>> d = IfThenElse(this.x > 0, VarInt, Byte)
>>> d.build(255, dict(x=1))
b'\xff\x01'
>>> d.build(255, dict(x=0))
b'\xff'
```

exception construct.core.**IndexFieldError**

class construct.core.**Indexing**(*subcon*, *count*, *index*, *empty=None*)

Adapter for indexing a list (getting a single item from that list). Works with Range and Sequence and their lazy equivalents.

Parameters

- **subcon** – Construct instance, subcon to index
- **count** – integer, expected number of elements, needed during building
- **index** – integer, index of the list to get
- **empty** – object, value to fill the list with, during building

Example:

```
d = Indexing(Array(4,Byte), 4, 2, empty=0)
assert d.parse(b"\x01\x02\x03\x04") == 3
assert d.build(3) == b"\x00\x00\x03\x00"
assert d.sizeof() == 4
```

exception construct.core.**IntegerError**

class construct.core.**Lazy**(*subcon*)

Lazyfies a field.

This wrapper allows you to do lazy parsing of individual fields inside a normal Struct (without using LazyStruct which may not work in every scenario). It is also used by KaitaiStruct compiler to emit *instances* because those are not processed greedily, and they may refer to other not yet parsed fields. Those are 2 entirely different applications but semantics are the same.

Parsing saves the current stream offset and returns a lambda. If and when that lambda gets evaluated, it seeks the stream to then-current position, parses the subcon, and seeks the stream back to previous position. Building evaluates that lambda into an object (if needed), then defers to subcon. Size also defers to subcon.

Parameters **subcon** – Construct instance

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- ***StreamError*** – stream is not seekable and tellable

Example:

```
>>> d = Lazy(Byte)
>>> x = d.parse(b'\x00')
>>> x
<function construct.core.Lazy._parse.<locals>.execute>
>>> x()
0
>>> d.build(0)
b'\x00'
>>> d.build(x)
b'\x00'
>>> d.sizeof()
1
```

class construct.core.**LazyArray**(*count*, *subcon*)

Equivalent to *Array*, but the subcon is not parsed when possible (it gets skipped if the size can be measured by _actualsize or _sizeof method). See its docstring for details.

Fields are parsed depending on some factors:

- Some fields like Int* Float* Bytes(5) Array(5,Byte) Pointer are fixed-size and are therefore skipped. Stream is not read.
- Some fields like Bytes(this.field) are variable-size but their size is known during parsing when there is a corresponding context entry. Those fields are also skipped. Stream is not read.
- Some fields like Prefixed PrefixedArray PascalString are variable-size but their size can be computed by partially reading the stream. Only first few bytes are read (the lengthfield).
- Other fields like VarInt need to be parsed. Stream position that is left after the field was parsed is used.
- Some fields may not work properly, due to the fact that this class attempts to skip fields, and parses them only out of necessity. Miscellaneous fields often have size defined as 0, and fixed sized fields are skippable.

Note there are restrictions:

- If a field references another field within inner (nested) or outer (super) struct, things may break. Context is nested, but this class was not rigorously tested in that manner.

Building and sizeof are greedy, like in Array.

Parameters

- **count** – integer or context lambda, strict amount of elements
- **subcon** – Construct instance, subcon to process individual elements

class construct.core.**LazyBound**(*subconfunc*)

Field that binds to the subcon only at runtime (during parsing and building, not ctor). Useful for recursive data structures, like linked-lists and trees, where a construct needs to refer to itself (while it does not exist yet in the namespace).

Note that it is possible to obtain same effect without using this class, using a loop. However there are usecases where that is not possible (if remaining nodes cannot be sized-up, and there is data following the recursive structure). There is also a significant difference, namely that LazyBound actually does greedy parsing while the loop does lazy parsing. See examples.

To break recursion, use *If* field. See examples.

Parameters **subconfunc** – parameter-less lambda returning Construct instance, can also return itself

Example:

```
d = Struct(
    "value" / Byte,
    "next" / If(this.value > 0, LazyBound(lambda: d)),
)
>>> print(d.parse(b"\x05\x09\x00"))
Container:
    value = 5
    next = Container:
        value = 9
        next = Container:
            value = 0
            next = None
```

```
d = Struct(
    "value" / Byte,
    "next" / GreedyBytes,
)
data = b"\x05\x09\x00"
while data:
    x = d.parse(data)
    data = x.next
    print(x)
# print outputs
Container:
    value = 5
    next = \t\x00 (total 2)
# print outputs
Container:
    value = 9
    next = \x00 (total 1)
# print outputs
Container:
    value = 0
    next =  (total 0)
```

class construct.core.**LazyContainer**(*struct*, *stream*, *offsets*, *values*, *context*, *path*)
Used internally.

items() → list of D's (key, value) pairs, as 2-tuples

keys() → list of D's keys

values() → list of D's values

class construct.core.**LazyListContainer**(*subcon*, *stream*, *count*, *offsets*, *values*, *context*, *path*)
Used internally.

class construct.core.**LazyStruct**(**subcons*, ***subconskw*)
Equivalent to *Struct*, but when this class is parsed, most fields are not parsed (they are skipped if their size can be measured by _actualsize or _sizeof method). See its docstring for details.

Fields are parsed depending on some factors:

- Some fields like Int* Float* Bytes(5) Array(5,Byte) Pointer are fixed-size and are therefore skipped. Stream is not read.
- Some fields like Bytes(this.field) are variable-size but their size is known during parsing when there is a corresponding context entry. Those fields are also skipped. Stream is not read.

- Some fields like Prefixed PrefixedArray PascalString are variable-size but their size can be computed by partially reading the stream. Only first few bytes are read (the lengthfield).
- Other fields like VarInt need to be parsed. Stream position that is left after the field was parsed is used.
- Some fields may not work properly, due to the fact that this class attempts to skip fields, and parses them only out of necessity. Miscellaneous fields often have size defined as 0, and fixed sized fields are skippable.

Note there are restrictions:

- If a field like Bytes(this.field) references another field in the same struct, you need to access the referenced field first (to trigger its parsing) and then you can access the Bytes field. Otherwise it would fail due to missing context entry.
- If a field references another field within inner (nested) or outer (super) struct, things may break. Context is nested, but this class was not rigorously tested in that manner.

Building and sizeof are greedy, like in Struct.

Parameters

- ***subcons** – Construct instances, list of members, some can be anonymous
- ****subconskw** – Construct instances, list of members (requires Python 3.6)

class construct.core.**Mapping**(*subcon*, *mapping*)

Adapter that maps objects to other objects. Translates objects after parsing and before building. Can for example, be used to translate between enum34 objects and strings, but Enum class supports enum34 already and is recommended.

Parameters

- **subcon** – Construct instance
- **mapping** – dict, for encoding (building) mapping, the reversed is used for parsing mapping

Raises ***MappingError*** – parsing or building but no mapping found

Example:

```
>>> x = object
>>> d = Mapping(Byte, {x:0})
>>> d.parse(b"\x00")
x
>>> d.build(x)
b'\x00'
```

exception construct.core.**MappingError**

class construct.core.**NamedTuple**(*tuplename*, *tuplefields*, *subcon*)

Both arrays, structs, and sequences can be mapped to a namedtuple from collections module. To create a named tuple, you need to provide a name and a sequence of fields, either a string with space-separated names or a list of string names, like the standard namedtuple.

Parses into a collections.namedtuple instance, and builds from such instance (although it also builds from lists and dicts). Size is undefined.

Parameters

- **tuplename** – string
- **tuplefields** – string or list of strings
- **subcon** – Construct instance, either Struct Sequence Array GreedyRange

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- ***NamedTupleError*** – subcon is neither Struct Sequence Array GreedyRange

Can propagate collections exceptions.

Example:

```
>>> d = NamedTuple("coord", "x y z", Byte[3])
>>> d = NamedTuple("coord", "x y z", Byte >> Byte >> Byte)
>>> d = NamedTuple("coord", "x y z", "x"/Byte + "y"/Byte + "z"/Byte)
>>> d.parse(b"123")
coord(x=49, y=50, z=51)
```

exception construct.core.**NamedTupleError**

construct.core.**NoneOf**(*subcon*, *invalids*)

Validates that the object is none of the listed values, both during parsing and building.

Note: For performance, *valids* should be a set/frozenset.

Parameters

- **subcon** – Construct instance, subcon to validate
- **invalids** – collection implementing __contains__, usually a list or set

Raises ***ValidationError*** – parsed or build value is among invalids

class construct.core.**NullStripped**(*subcon*, *pad='x00'*)

Restricts parsing to bytes except padding left of EOF.

Parsing reads entire stream, then strips the data from right to left of null bytes, then parses subcon using new BytesIO made of said data. Building defers to subcon as-is. Size is undefined, because it reads till EOF.

The pad can be multiple bytes, to support string classes with UTF16/32 encodings.

Parameters

- **subcon** – Construct instance
- **pad** – optional, bytes, padding byte-string, default is x00 single null byte

Raises ***PaddingError*** – pad is less than 1 bytes in length

Example:

```
>>> d = NullStripped(Byte)
>>> d.parse(b'\xff\x00\x00')
255
>>> d.build(255)
b'\xff'
```

class construct.core.**NullTerminated**(*subcon*, *term='x00'*, *include=False*, *consume=True*, *require=True*)

Restricts parsing to bytes preceding a null byte.

Parsing reads one byte at a time and accumulates it with previous bytes. When term was found, (by default) consumes but discards the term. When EOF was found, (by default) raises same StreamError exception. Then

subcon is parsed using new BytesIO made with said data. Building builds the subcon and then writes the term. Size is undefined.

The term can be multiple bytes, to support string classes with UTF16/32 encodings.

Parameters

- **subcon** – Construct instance
- **term** – optional, bytes, terminator byte-string, default is x00 single null byte
- **include** – optional, bool, if to include terminator in resulting data, default is False
- **consume** – optional, bool, if to consume terminator or leave it in the stream, default is True
- **require** – optional, bool, if EOF results in failure or not, default is True

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- ***StreamError*** – encountered EOF but require is not disabled
- ***PaddingError*** – terminator is less than 1 bytes in length

Example:

```
>>> d = NullTerminated(Byte)
>>> d.parse(b'\xff\x00')
255
>>> d.build(255)
b'\xff\x00'
```

construct.core.**OneOf**(*subcon*, *valids*)

Validates that the object is one of the listed values, both during parsing and building.

Note: For performance, *valids* should be a set/frozenset.

Parameters

- **subcon** – Construct instance, subcon to validate
- **valids** – collection implementing __contains__, usually a list or set

Raises ***ValidationError*** – parsed or build value is not among valids

Example:

```
>>> d = OneOf(Byte, [1,2,3])
>>> d.parse(b"\x01")
1
>>> d.parse(b"\xff")
construct.core.ValidationError: object failed validation: 255
```

construct.core.**Optional**(*subcon*)

Makes an optional field.

Parsing attempts to parse subcon. If sub-parsing fails, returns None and reports success. Building attempts to build subcon. If sub-building fails, writes nothing and reports success. Size is undefined, because whether bytes would be consumed or produced depends on actual data and actual context.

Parameters **subcon** – Construct instance

Example:

```
Optional  <-->  Select(subcon, Pass)

>>> d = Optional(Int64ul)
>>> d.parse(b"12345678")
4050765991979987505
>>> d.parse(b"")
None
>>> d.build(1)
b'\x01\x00\x00\x00\x00\x00\x00\x00'
>>> d.build(None)
b''
```

class construct.core.**Padded**(*length*, *subcon*, *pattern='x00'*)

Appends additional null bytes to achieve a length.

Parsing first parses the subcon, then uses stream.tell() to measure how many bytes were read and consumes additional bytes accordingly. Building first builds the subcon, then uses stream.tell() to measure how many bytes were written and produces additional bytes accordingly. Size is same as *length*, but negative amount results in error. Note that subcon can actually be variable size, it is the eventual amount of bytes that is read or written during parsing or building that determines actual padding.

Parameters

- **length** – integer or context lambda, length of the padding
- **subcon** – Construct instance
- **pattern** – optional, b-character, padding pattern, default is \x00

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- ***PaddingError*** – length is negative
- ***PaddingError*** – subcon read or written more than the length (would cause negative pad)
- ***PaddingError*** – pattern is not bytes of length 1

Can propagate any exception from the lambda, possibly non-ConstructError.

Example:

```
>>> d = Padded(4, Byte)
>>> d.build(255)
b'\xff\x00\x00\x00'
>>> d.parse(_)
255
>>> d.sizeof()
4

>>> d = Padded(4, VarInt)
>>> d.build(1)
b'\x01\x00\x00\x00'
>>> d.build(70000)
b'\xf0\xa2\x04\x00'
```

construct.core.**PaddedString**(*length*, *encoding*)
Configurable, fixed-length or variable-length string field.

When parsing, the byte string is stripped of null bytes (per encoding unit), then decoded. Length is an integer or context lambda. When building, the string is encoded and then padded to specified length. If encoded string is larger than the specified length, it fails with PaddingError. Size is same as length parameter.

Warning: PaddedString and CString only support encodings explicitly listed in *possiblestringencodings*.

Parameters

- **length** – integer or context lambda, length in bytes (not unicode characters)
- **encoding** – string like: utf8 utf16 utf32 ascii

Raises

- ***StringError*** – building a non-unicode string
- ***StringError*** – selected encoding is not on supported list

Can propagate any exception from the lambda, possibly non-ConstructError.

Example:

```
>>> d = PaddedString(10, "utf8")
>>> d.build(u"")
b'\xd0\x90\xd1\x84\xd0\xbe\xd0\xbd\x00\x00'
>>> d.parse(_)
u''
```

construct.core.**Padding**(*length*, *pattern='\x00'*)
Appends null bytes.

Parsing consumes specified amount of bytes and discards it. Building writes specified pattern byte multiplied into specified length. Size is same as specified.

Parameters

- **length** – integer or context lambda, length of the padding
- **pattern** – b-character, padding pattern, default is \x00

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- ***PaddingError*** – length was negative
- ***PaddingError*** – pattern was not bytes (b-character)

Can propagate any exception from the lambda, possibly non-ConstructError.

Example:

```
>>> d = Padding(4) or Padded(4, Pass)
>>> d.build(None)
b'\x00\x00\x00\x00'
>>> d.parse(b"****")
```

(continues on next page)

(continued from previous page)

```
None
>>> d.sizeof()
4
```

exception construct.core.**PaddingError**

construct.core.**PascalString**(*lengthfield*, *encoding*)

Length-prefixed string. The length field can be variable length (such as VarInt) or fixed length (such as Int64ub). VarInt is recommended when designing new protocols. Stored length is in bytes, not characters. Size is not defined.

Parameters

- **lengthfield** – Construct instance, field used to parse and build the length (like VarInt Int64ub)
- **encoding** – string like: utf8 utf16 utf32 ascii

Raises ***StringError*** – building a non-unicode string

Example:

```
>>> d = PascalString(VarInt, "utf8")
>>> d.build(u"")
b'\x08\xd0\x90\xd1\x84\xd0\xbe\xd0\xbd'
>>> d.parse(_)
u''
```

class construct.core.**Peek**(*subcon*)

Peeks at the stream.

Parsing sub-parses (and returns None if failed), then reverts stream to original position. Building does nothing (its NOT deferred). Size is defined as 0 because there is no building.

This class is used in *Union* class to parse each member.

Parameters **subcon** – Construct instance

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- ***StreamError*** – stream is not seekable and tellable

Example:

```
>>> d = Sequence(Peek(Int8ub), Peek(Int16ub))
>>> d.parse(b"\x01\x02")
[1, 258]
>>> d.sizeof()
0
```

class construct.core.**Pointer**(*offset*, *subcon*, *stream=None*)

Jumps in the stream forth and back for one field.

Parsing and building seeks the stream to new location, processes subcon, and seeks back to original location. Size is defined as 0 but that does not mean no bytes are written into the stream.

Offset can be positive, indicating a position from stream beginning forward, or negative, indicating a position from EOF backwards.

Parameters

- **offset** – integer or context lambda, positive or negative
- **subcon** – Construct instance
- **stream** – None to use original stream (default), or context lambda to provide a different stream

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- ***StreamError*** – stream is not seekable and tellable

Can propagate any exception from the lambda, possibly non-ConstructError.

Example:

```
>>> d = Pointer(8, Bytes(1))
>>> d.parse(b"abcdefghijkl")
b'i'
>>> d.build(b"Z")
b'\x00\x00\x00\x00\x00\x00\x00\x00Z'
```

class construct.core.**Prefixed**(*lengthfield*, *subcon*, *includelength=False*)

Prefixes a field with byte count.

Parses the length field. Then reads that amount of bytes, and parses subcon using only those bytes. Constructs that consume entire remaining stream are constrained to consuming only the specified amount of bytes (a substream). When building, data gets prefixed by its length. Optionally, length field can include its own size. Size is the sum of both fields sizes, unless either raises SizeofError.

Analog to *PrefixedArray* which prefixes with an element count, instead of byte count. Semantics is similar but implementation is different.

VarInt is recommended for new protocols, as it is more compact and never overflows.

Parameters

- **lengthfield** – Construct instance, field used for storing the length
- **subcon** – Construct instance, subcon used for storing the value
- **includelength** – optional, bool, whether length field should include its own size, default is False

Raises ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes

Example:

```
>>> d = Prefixed(VarInt, GreedyRange(Int32ul))
>>> d.parse(b"\x08abcdefgh")
[1684234849, 1751606885]

>>> d = PrefixedArray(VarInt, Int32ul)
>>> d.parse(b"\x02abcdefgh")
[1684234849, 1751606885]
```

construct.core.**PrefixedArray**(*countfield*, *subcon*)

Prefixes an array with item count (as opposed to prefixed by byte count, see *Prefixed*).

VarInt is recommended for new protocols, as it is more compact and never overflows.

Parameters

- **countfield** – Construct instance, field used for storing the element count
- **subcon** – Construct instance, subcon used for storing each element

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- ***RangeError*** – consumed or produced too little elements

Example:

```
>>> d = Prefixed(VarInt, GreedyRange(Int32ul))
>>> d.parse(b"\x08abcdefgh")
[1684234849, 1751606885]

>>> d = PrefixedArray(VarInt, Int32ul)
>>> d.parse(b"\x02abcdefgh")
[1684234849, 1751606885]
```

class construct.core.**ProcessRotateLeft**(*amount*, *group*, *subcon*)

Transforms bytes between the underlying stream and the subcon.

Used internally by KaitaiStruct compiler, when translating *process: rol/ror* tags.

Parsing reads till EOF, rotates (shifts) the data *left* by amount in bits, then feeds that data into subcon. Building first builds the subcon into separate BytesIO stream, rotates *right* by negating amount, then writes that data into the main stream. Size is the same as subcon, unless it raises SizeofError.

Parameters

- **amount** – integer or context lambda, shift by this amount in bits, treated modulo (group x 8)
- **group** – integer or context lambda, shifting is applied to chunks of this size in bytes
- **subcon** – Construct instance

Raises

- ***RotationError*** – group is less than 1
- ***RotationError*** – data length is not a multiple of group size

Can propagate any exception from the lambda, possibly non-ConstructError.

Example:

```
>>> d = ProcessRotateLeft(4, 1, Int16ub)
>>> d.parse(b'\x0f\xf0')
0xf00f
>>> d = ProcessRotateLeft(4, 2, Int16ub)
>>> d.parse(b'\x0f\xf0')
0xff00
>>> d.sizeof()
2
```

class construct.core.**ProcessXor**(*padfunc*, *subcon*)

Transforms bytes between the underlying stream and the subcon.

Used internally by KaitaiStruct compiler, when translating *process: xor* tags.

Parsing reads till EOF, xors data with the pad, then feeds that data into subcon. Building first builds the subcon into separate BytesIO stream, xors data with the pad, then writes that data into the main stream. Size is the same as subcon, unless it raises SizeofError.

Parameters

- **padfunc** – integer or bytes or context lambda, single or multiple bytes to xor data with
- **subcon** – Construct instance

Raises ***StringError*** – pad is not integer or bytes

Can propagate any exception from the lambda, possibly non-ConstructError.

Example:

```
>>> d = ProcessXor(0xf0 or b'\xf0', Int16ub)
>>> d.parse(b"\x00\xff")
0xf00f
>>> d.sizeof()
2
```

exception construct.core.**RangeError**

class construct.core.**RawCopy**(*subcon*)

Used to obtain byte representation of a field (aside of object value).

Returns a dict containing both parsed subcon value, the raw bytes that were consumed by subcon, starting and ending offset in the stream, and amount in bytes. Builds either from raw bytes representation or a value used by subcon. Size is same as subcon.

Object is a dictionary with either "data" or "value" keys, or both.

Parameters **subcon** – Construct instance

Raises

- ***StreamError*** – stream is not seekable and tellable
- ***RawCopyError*** – building and neither data or value was given
- ***StringError*** – building from non-bytes value, perhaps unicode

Example:

```
>>> d = RawCopy(Byte)
>>> d.parse(b"\xff")
Container(data=b'\xff')(value=255)(offset1=0)(offset2=1)(length=1)
>>> d.build(dict(data=b"\xff"))
'\xff'
>>> d.build(dict(value=255))
'\xff'
```

exception construct.core.**RawCopyError**

class construct.core.**Rebuffered**(*subcon*, *tailcutoff=None*)

Caches bytes from underlying stream, so it becomes seekable and tellable, and also becomes blocking on reading. Useful for processing non-file streams like pipes, sockets, etc.

> **Warning:** Experimental implementation. May not be mature enough.

Parameters

- **subcon** – Construct instance, subcon which will operate on the buffered stream
- **tailcutoff** – optional, integer, amount of bytes kept in buffer, by default buffers everything

Can also raise arbitrary exceptions in its implementation.

Example:

```
Rebuffered(..., tailcutoff=1024).parse_stream(nonseekable_stream)
```

class construct.core.**Rebuild**(*subcon*, *func*)

Field where building does not require a value, because the value gets recomputed when needed. Comes handy when building a Struct from a dict with missing keys. Useful for length and count fields when *Prefixed* and *PrefixedArray* cannot be used.

Parsing defers to subcon. Building is defered to subcon, but it builds from a value provided by the context lambda (or constant). Size is the same as subcon, unless it raises SizeofError.

Difference between Default and Rebuild, is that in first the build value is optional and in second the build value is ignored.

Parameters

- **subcon** – Construct instance
- **func** – context lambda or constant value

Raises ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes

Can propagate any exception from the lambda, possibly non-ConstructError.

Example:

```
>>> d = Struct(
...     "count" / Rebuild(Byte, len_(this.items)),
...     "items" / Byte[this.count],
... )
>>> d.build(dict(items=[1,2,3]))
b'\x03\x01\x02\x03'
```

class construct.core.**Renamed**(*subcon*, *newname=None*, *newdocs=None*, *newparsed=None*)

Special wrapper that allows a Struct (or other similar class) to see a field as having a name (or a different name) or having a parsed hook. Library classes do not have names (its None). Renamed does not change a field, only wraps it like a candy with a label. Used internally by / and * operators.

Also this wrapper is responsible for building a path info (a chain of names) that gets attached to error message when parsing, building, or sizeof fails. Fields that are not named do not appear in the path string.

Parsing building and size are deferred to subcon.

Parameters

- **subcon** – Construct instance
- **newname** – optional, string
- **newdocs** – optional, string
- **newparsed** – optional, lambda

Example:

```
>>> "number" / Int32ub
<Renamed: number>
```

exception construct.core.**RepeatError**

class construct.core.**RepeatUntil**(*predicate*, *subcon*, *discard=False*)

Homogenous array of elements, similar to C# generic IEnumerable<T>, that repeats until the predicate indicates it to stop. Note that the last element (that predicate indicated as True) is included in the return list.

Parse iterates indefinately until last element passed the predicate. Build iterates indefinately over given list, until an element passed the precicate (or raises RepeatError if no element passed it). Size is undefined.

Parameters

- **predicate** – lambda that takes (obj, list, context) and returns True to break or False to continue (or a truthy value)
- **subcon** – Construct instance, subcon used to parse and build each element
- **discard** – optional, bool, if set then parsing returns empty list

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- ***RepeatError*** – consumed all elements in the stream but neither passed the predicate

Can propagate any exception from the lambda, possibly non-ConstructError.

Example:

```
>>> d = RepeatUntil(lambda x,lst,ctx: x > 7, Byte)
>>> d.build(range(20))
b'\x00\x01\x02\x03\x04\x05\x06\x07\x08'
>>> d.parse(b"\x01\xff\x02")
[1, 255]

>>> d = RepeatUntil(lambda x,lst,ctx: lst[-2:] == [0,0], Byte)
>>> d.parse(b"\x01\x00\x00\xff")
[1, 0, 0]
```

class construct.core.**RestreamData**(*datafunc*, *subcon*)

Parses a field on external data (but does not build).

Parsing defers to subcon, but provides it a separate BytesIO stream based on data provided by datafunc (a bytes literal or another BytesIO stream or Construct instances that returns bytes or context lambda). Building does nothing. Size is 0 because as far as other fields see it, this field does not produce or consume any bytes from the stream.

Parameters

- **datafunc** – bytes or BytesIO or Construct instance (that parses into bytes) or context lambda, provides data for subcon to parse from
- **subcon** – Construct instance

Can propagate any exception from the lambdas, possibly non-ConstructError.

Example:

```
>>> d = RestreamData(b"\x01", Int8ub)
>>> d.parse(b"")
1
>>> d.build(0)
b''

>>> d = RestreamData(NullTerminated(GreedyBytes), Int16ub)
>>> d.parse(b"\x01\x02\x00")
0x0102
>>> d = RestreamData(FixedSized(2, GreedyBytes), Int16ub)
>>> d.parse(b"\x01\x02\x00")
0x0102
```

class construct.core.**Restreamed**(*subcon*, *decoder*, *decoderunit*, *encoder*, *encoderunit*, *sizecomputer*)

Transforms bytes between the underlying stream and the (variable-sized) subcon.

Used internally to implement *Bitwise* *Bytewise* *ByteSwapped* *BitsSwapped*.

Warning: Remember that subcon must consume or produce an amount of bytes that is a multiple of encoding or decoding units. For example, in a Bitwise context you should process a multiple of 8 bits or the stream will fail during parsing/building.

Warning: Do NOT use seeking/telling classes inside Restreamed context.

Parameters

- **subcon** – Construct instance
- **decoder** – bytes-to-bytes function, used on data chunks when parsing
- **decoderunit** – integer, decoder takes chunks of this size
- **encoder** – bytes-to-bytes function, used on data chunks when building
- **encoderunit** – integer, encoder takes chunks of this size
- **sizecomputer** – function that computes amount of bytes outputed

Can propagate any exception from the lambda, possibly non-ConstructError. Can also raise arbitrary exceptions in RestreamedBytesIO implementation.

Example:

```
Bitwise  <--> Restreamed(subcon, bits2bytes, 8, bytes2bits, 1, lambda n: n//8)
Bytewise <--> Restreamed(subcon, bytes2bits, 1, bits2bytes, 8, lambda n: n*8)
```

exception construct.core.**RotationError**

class construct.core.**Seek**(*at*, *whence=0*)

Seeks the stream.

Parsing and building seek the stream to given location (and whence), and return stream.seek() return value. Size is not defined.

See also:

Analog *Pointer* wrapper that has same side effect but also processes a subcon, and also seeks back.

Parameters

- **at** – integer or context lambda, where to jump to
- **whence** – optional, integer or context lambda, is the offset from beginning (0) or from current position (1) or from EOF (2), default is 0

Raises ***StreamError*** – stream is not seekable

Can propagate any exception from the lambda, possibly non-ConstructError.

Example:

```
>>> d = (Seek(5) >> Byte)
>>> d.parse(b"01234x")
[5, 120]

>>> d = (Bytes(10) >> Seek(5) >> Byte)
>>> d.build([b"0123456789", None, 255])
b'01234\xff6789'
```

class construct.core.**Select**(**subcons*, ***subconskw*)

Selects the first matching subconstruct.

Parses and builds by literally trying each subcon in sequence until one of them parses or builds without exception. Stream gets reverted back to original position after each failed attempt, but not if parsing succeeds. Size is not defined.

Parameters

- ***subcons** – Construct instances, list of members, some can be anonymous
- ****subconskw** – Construct instances, list of members (requires Python 3.6)

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- ***StreamError*** – stream is not seekable and tellable
- ***SelectError*** – neither subcon succeded when parsing or building

Example:

```
>>> d = Select(Int32ub, CString("utf8"))
>>> d.build(1)
b'\x00\x00\x00\x01'
>>> d.build(u"")
b'\xd0\x90\xd1\x84\xd0\xbe\xd0\xbd\x00'

Alternative syntax, but requires Python 3.6 or any PyPy:
>>> Select(num=Int32ub, text=CString("utf8"))
```

exception construct.core.**SelectError**

class construct.core.**Sequence**(**subcons*, ***subconskw*)

Sequence of usually un-named constructs. The members are parsed and build in the order they are defined. If a member is named, its parsed value gets inserted into the context. This allows using members that refer to previous members. *Embedded* fields do not need to (and should not) be named.

Operator >> can also be used to make Sequences (although not recommended).

Parses into a ListContainer (list with pretty-printing) where values are in same order as subcons. Builds from a list (not necessarily a ListContainer) where each subcon is given the element at respective position. Size is the sum of all subcon sizes, unless any subcon raises SizeofError.

This class does context nesting, meaning its members are given access to a new dictionary where the "_" entry points to the outer context. When parsing, each member gets parsed and subcon parse return value is inserted into context under matching key only if the member was named. When building, the matching entry gets inserted into context before subcon gets build, and if subcon build returns a new value (not None) that gets replaced in the context.

This class supports embedding. *Embedded* semantics dictate, that during instance creation (in ctor), each field is checked for embedded flag, and its subcon members are merged. This changes behavior of some code examples. Only few classes are supported: Struct Sequence FocusedSeq Union LazyStruct, although those can be used interchangably (a Struct can embed a Sequence, or rather its members).

This class exposes subcons as attributes. You can refer to subcons that were inlined (and therefore do not exist as variable in the namespace) by accessing the struct attributes, under same name. Also note that compiler does not support this feature. See examples.

This class exposes subcons in the context. You can refer to subcons that were inlined (and therefore do not exist as variable in the namespace) within other inlined fields using the context. Note that you need to use a lambda (*this* expression is not supported). Also note that compiler does not support this feature. See examples.

This class supports stopping. If *StopIf* field is a member, and it evaluates its lambda as positive, this class ends parsing or building as successful without processing further fields.

Parameters

- ***subcons** – Construct instances, list of members, some can be named
- ****subconskw** – Construct instances, list of members (requires Python 3.6)

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- **KeyError** – building a subcon but found no corresponding key in dictionary

Example:

```
>>> d = Sequence(Byte, Float32b)
>>> d.build([0, 1.23])
b'\x00?\x9dp\xa4'
>>> d.parse(_)
[0, 1.2300000190734863] # a ListContainer

>>> d = Sequence(
...     "animal" / Enum(Byte, giraffe=1),
... )
>>> d.animal.giraffe
'giraffe'
>>> d = Sequence(
...     "count" / Byte,
...     "data" / Bytes(lambda this: this.count - this._subcons.count.sizeof()),
... )
>>> d.build([3, b"12"])
b'\x0312'

Alternative syntax (not recommended):
>>> (Byte >> "Byte >> "c"/Byte >> "d"/Byte)
```

(continues on next page)

(continued from previous page)

```
Alternative syntax, but requires Python 3.6 or any PyPy:
>>> Sequence(a=Byte, b=Byte, c=Byte, d=Byte)
```

exception construct.core.**SizeofError**

class construct.core.**Slicing**(*subcon*, *count*, *start*, *stop*, *step=1*, *empty=None*)

Adapter for slicing a list. Works with GreedyRange and Sequence.

Parameters

- **subcon** – Construct instance, subcon to slice
- **count** – integer, expected number of elements, needed during building
- **start** – integer for start index (or None for entire list)
- **stop** – integer for stop index (or None for up-to-end)
- **step** – integer, step (or 1 for every element)
- **empty** – object, value to fill the list with, during building

Example:

```
d = Slicing(Array(4,Byte), 4, 1, 3, empty=0)
assert d.parse(b"\x01\x02\x03\x04") == [2,3]
assert d.build([2,3]) == b"\x00\x02\x03\x00"
assert d.sizeof() == 4
```

exception construct.core.**StopFieldError**

class construct.core.**StopIf**(*condfunc*)

Checks for a condition, and stops certain classes (*Struct* *Sequence* *GreedyRange*) from parsing or building further.

Parsing and building check the condition, and raise StopFieldError if indicated. Size is undefined.

Parameters **condfunc** – bool or context lambda (or truthy value)

Raises ***StopFieldError*** – used internally

Can propagate any exception from the lambda, possibly non-ConstructError.

Example:

```
>>> Struct('x'/Byte, StopIf(this.x == 0), 'y'/Byte)
>>> Sequence('x'/Byte, StopIf(this.x == 0), 'y'/Byte)
>>> GreedyRange(FocusedSeq(0, 'x'/Byte, StopIf(this.x == 0)))
```

exception construct.core.**StreamError**

class construct.core.**StringEncoded**(*subcon*, *encoding*)

Used internally.

exception construct.core.**StringError**

class construct.core.**Struct**(**subcons*, ***subconskw*)

Sequence of usually named constructs, similar to structs in C. The members are parsed and build in the order they are defined. If a member is anonymous (its name is None) then it gets parsed and the value discarded, or it gets build from nothing (from None).

Some fields do not need to be named, since they are built without value anyway. See: Const Padding Check Error Pass Terminated Seek Tell for examples of such fields. *Embedded* fields do not need to (and should not) be named.

Operator + can also be used to make Structs (although not recommended).

Parses into a Container (dict with attribute and key access) where keys match subcon names. Builds from a dict (not necessarily a Container) where each member gets a value from the dict matching the subcon name. If field has build-from-none flag, it gets build even when there is no matching entry in the dict. Size is the sum of all subcon sizes, unless any subcon raises SizeofError.

This class does context nesting, meaning its members are given access to a new dictionary where the "_" entry points to the outer context. When parsing, each member gets parsed and subcon parse return value is inserted into context under matching key only if the member was named. When building, the matching entry gets inserted into context before subcon gets build, and if subcon build returns a new value (not None) that gets replaced in the context.

This class supports embedding. *Embedded* semantics dictate, that during instance creation (in ctor), each field is checked for embedded flag, and its subcon members are merged. This changes behavior of some code examples. Only few classes are supported: Struct Sequence FocusedSeq Union LazyStruct, although those can be used interchangably (a Struct can embed a Sequence, or rather its members).

This class exposes subcons as attributes. You can refer to subcons that were inlined (and therefore do not exist as variable in the namespace) by accessing the struct attributes, under same name. Also note that compiler does not support this feature. See examples.

This class exposes subcons in the context. You can refer to subcons that were inlined (and therefore do not exist as variable in the namespace) within other inlined fields using the context. Note that you need to use a lambda (*this* expression is not supported). Also note that compiler does not support this feature. See examples.

This class supports stopping. If *StopIf* field is a member, and it evaluates its lambda as positive, this class ends parsing or building as successful without processing further fields.

Parameters

- ***subcons** – Construct instances, list of members, some can be anonymous
- ****subconskw** – Construct instances, list of members (requires Python 3.6)

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- **KeyError** – building a subcon but found no corresponding key in dictionary

Example:

```
>>> d = Struct("num"/Int8ub, "data"/Bytes(this.num))
>>> d.parse(b"\x04DATA")
Container(num=4)(data=b"DATA")
>>> d.build(dict(num=4, data=b"DATA"))
b"\x04DATA"

>>> d = Struct(Const(b"MZ"), Padding(2), Pass, Terminated)
>>> d.build({})
b'MZ\x00\x00'
>>> d.parse(_)
Container()
>>> d.sizeof()
4
```

(continues on next page)

(continued from previous page)

```
>>> d = Struct(
...     "animal" / Enum(Byte, giraffe=1),
... )
>>> d.animal.giraffe
'giraffe'
>>> d = Struct(
...     "count" / Byte,
...     "data" / Bytes(lambda this: this.count - this._subcons.count.sizeof()),
... )
>>> d.build(dict(count=3, data=b"12"))
b'\x0312'

Alternative syntax (not recommended):
>>> ("a"/Byte + "b"/Byte + "c"/Byte + "d"/Byte)

Alternative syntax, but requires Python 3.6 or any PyPy:
>>> Struct(a=Byte, b=Byte, c=Byte, d=Byte)
```

class construct.core.**Subconstruct**(*subcon*)

Abstract subconstruct (wraps an inner construct, inheriting its name and flags). Parsing and building is by default deferred to subcon, same as sizeof.

Parameters **subcon** – Construct instance

class construct.core.**Switch**(*keyfunc*, *cases*, *default=None*)

A conditional branch.

Parsing and building evaluate keyfunc and select a subcon based on the value and dictionary entries. Dictionary (cases) maps values into subcons. If no case matches then *default* is used (that is Pass by default). Note that *default* is a Construct instance, not a dictionary key. Size is evaluated in same way as parsing and building, by evaluating keyfunc and selecting a field accordingly.

Parameters

- **keyfunc** – context lambda or constant, that matches some key in cases
- **cases** – dict mapping keys to Construct instances
- **default** – optional, Construct instance, used when keyfunc is not found in cases, Pass is default value for this parameter, Error is a possible value for this parameter

Raises ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes

Can propagate any exception from the lambda, possibly non-ConstructError.

Example:

```
>>> d = Switch(this.n, { 1:Int8ub, 2:Int16ub, 4:Int32ub })
>>> d.build(5, n=1)
b'\x05'
>>> d.build(5, n=4)
b'\x00\x00\x00\x05'

>>> d = Switch(this.n, {}, default=Byte)
>>> d.parse(b"\x01", n=255)
1
>>> d.build(1, n=255)
b"\x01"
```

exception construct.core.**SwitchError**

class construct.core.**SymmetricAdapter**(*subcon*)
Abstract adapter class.

Needs to implement *_decode()* only, for both parsing and building.

Parameters **subcon** – Construct instance

exception construct.core.**TerminatedError**

construct.core.**Timestamp**(*subcon*, *unit*, *epoch*)
Datetime, represented as Arrow object.

Note that accuracy is not guaranteed, because building rounds the value to integer (even when Float subcon is used), due to floating-point errors in general, and because MSDOS scheme has only 5-bit (32 values) seconds field (seconds are rounded to multiple of 2).

Unit is a fraction of a second. 1 is second resolution, 10**-3 is milliseconds resolution, 10**-6 is microseconds resolution, etc. Usually its 1 on Unix and MacOSX, 10**-7 on Windows. Epoch is a year (if integer) or a specific day (if Arrow object). Usually its 1970 on Unix, 1904 on MacOSX, 1600 on Windows. MSDOS format doesnt support custom unit or epoch, it uses 2-seconds resolution and 1980 epoch.

Parameters

- **subcon** – Construct instance like Int* Float*, or Int32ub with msdos format
- **unit** – integer or float, or msdos string
- **epoch** – integer, or Arrow instance, or msdos string

Raises

- **ImportError** – arrow could not be imported during ctor
- ***TimestampError*** – subcon is not a Construct instance
- ***TimestampError*** – unit or epoch is a wrong type

Example:

```
>>> d = Timestamp(Int64ub, 1., 1970)
>>> d.parse(b'\x00\x00\x00\x00ZIz\x00')
<Arrow [2018-01-01T00:00:00+00:00]>
>>> d = Timestamp(Int32ub, "msdos", "msdos")
>>> d.parse(b'H9\x8c"')
<Arrow [2016-01-25T17:33:04+00:00]>
```

exception construct.core.**TimestampError**

class construct.core.**Transformed**(*subcon*, *decodefunc*, *decodeamount*, *encodefunc*, *encodeamount*)
Transforms bytes between the underlying stream and the (fixed-sized) subcon.

Parsing reads a specified amount (or till EOF), processes data using a bytes-to-bytes decoding function, then parses subcon using those data. Building does build subcon into separate bytes, then processes it using encoding bytes-to-bytes function, then writes those data into main stream. Size is reported as *decodeamount* or *encodeamount* if those are equal, otherwise its SizeofError.

Used internally to implement *Bitwise* *Bytewise* *ByteSwapped* *BitsSwapped* .

Possible use-cases include encryption, obfuscation, byte-level encoding.

Warning: Remember that subcon must consume (or produce) an amount of bytes that is same as *decodeamount* (or *encodeamount*).

Warning: Do NOT use seeking/telling classes inside Transformed context.

Parameters

- **subcon** – Construct instance
- **decodefunc** – bytes-to-bytes function, applied before parsing subcon
- **decodeamount** – integer, amount of bytes to read
- **encodefunc** – bytes-to-bytes function, applied after building subcon
- **encodeamount** – integer, amount of bytes to write

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- ***StreamError*** – subcon build and encoder transformed more or less than *encodeamount* bytes, if amount is specified
- ***StringError*** – building from non-bytes value, perhaps unicode

Can propagate any exception from the lambdas, possibly non-ConstructError.

Example:

```
>>> d = Transformed(Bytes(16), bytes2bits, 2, bits2bytes, 2)
>>> d.parse(b"\x00\x00")
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'

>>> d = Transformed(GreedyBytes, bytes2bits, None, bits2bytes, None)
>>> d.parse(b"\x00\x00")
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
```

class construct.core.**Tunnel**(*subcon*)

Abstract class that allows other constructs to read part of the stream as if they were reading the entrie stream. See Prefixed for example.

Needs to implement *_decode()* for parsing and *_encode()* for building.

class construct.core.**Union**(*parsefrom*, **subcons*, ***subconskw*)

Treats the same data as multiple constructs (similar to C union) so you can look at the data in multiple views. Fields are usually named (so parsed values are inserted into dictionary under same name). *Embedded* fields do not need to (and should not) be named.

Parses subcons in sequence, and reverts the stream back to original position after each subcon. Afterwards, advances the stream by selected subcon. Builds from first subcon that has a matching key in given dict. Size is undefined (because parsefrom is not used for building).

This class does context nesting, meaning its members are given access to a new dictionary where the "_" entry points to the outer context. When parsing, each member gets parsed and subcon parse return value is inserted into context under matching key only if the member was named. When building, the matching entry gets inserted

into context before subcon gets build, and if subcon build returns a new value (not None) that gets replaced in the context.

This class supports embedding. *Embedded* semantics dictate, that during instance creation (in ctor), each field is checked for embedded flag, and its subcon members are merged. This changes behavior of some code examples. Only few classes are supported: Struct Sequence FocusedSeq Union LazyStruct, although those can be used interchangably (a Struct can embed a Sequence, or rather its members).

This class exposes subcons as attributes. You can refer to subcons that were inlined (and therefore do not exist as variable in the namespace) by accessing the struct attributes, under same name. Also note that compiler does not support this feature. See examples.

This class exposes subcons in the context. You can refer to subcons that were inlined (and therefore do not exist as variable in the namespace) within other inlined fields using the context. Note that you need to use a lambda (*this* expression is not supported). Also note that compiler does not support this feature. See examples.

Warning: If you skip *parsefrom* parameter then stream will be left back at starting offset, not seeked to any common denominator.

Parameters

- **parsefrom** – how to leave stream after parsing, can be integer index or string name selecting a subcon, or None (leaves stream at initial offset, the default), or context lambda
- ***subcons** – Construct instances, list of members, some can be anonymous
- ****subconskw** – Construct instances, list of members (requires Python 3.6)

Raises

- ***StreamError*** – requested reading negative amount, could not read enough bytes, requested writing different amount than actual data, or could not write all bytes
- ***StreamError*** – stream is not seekable and tellable
- ***UnionError*** – selector does not match any subcon, or dict given to build does not contain any keys matching any subcon
- **IndexError** – selector does not match any subcon
- **KeyError** – selector does not match any subcon

Can propagate any exception from the lambda, possibly non-ConstructError.

Example:

```
>>> d = Union(0,
...     "raw" / Bytes(8),
...     "ints" / Int32ub[2],
...     "shorts" / Int16ub[4],
...     "chars" / Byte[8],
... )
>>> d.parse(b"12345678")
Container(raw=b'12345678', ints=[825373492, 892745528], shorts=[12594, 13108,␣
↪13622, 14136], chars=[49, 50, 51, 52, 53, 54, 55, 56])
>>> d.build(dict(chars=range(8)))
b'\x00\x01\x02\x03\x04\x05\x06\x07'

>>> d = Union(None,
...     "animal" / Enum(Byte, giraffe=1),
```

(continues on next page)

(continued from previous page)

```
... )
>>> d.animal.giraffe
'giraffe'
>>> d = Union(None,
...     "chars" / Byte[4],
...     "data" / Bytes(lambda this: this._subcons.chars.sizeof()),
... )
>>> d.parse(b"\x01\x02\x03\x04")
Container(chars=[1, 2, 3, 4])(data=b'\x01\x02\x03\x04')

Alternative syntax, but requires Python 3.6 or any PyPy:
>>> Union(0, raw=Bytes(8), ints=Int32ub[2], shorts=Int16ub[4], chars=Byte[8])
```

exception construct.core.**UnionError**

exception construct.core.**ValidationError**

class construct.core.**Validator**(*subcon*)
Abstract class that validates a condition on the encoded/decoded object.

Needs to implement *_validate()* that returns a bool (or a truthy value)

Parameters **subcon** – Construct instance

construct.core.**encodingunit**(*encoding*)
Used internally.

2.19 construct.lib – entire module

construct.lib.**bits2bytes**(*data*)
Converts between bit and byte representations in b-strings.

Example:

```
>>> bits2bytes(b"\x00\x01\x01\x00\x00\x00\x00\x01\x00\x01\x01\x00\x00\x00\x01\x00
↪")
b'ab'
```

construct.lib.**bits2integer**(*data*, *signed=False*)
Converts a bit-string into an integer. Set sign to interpret the number as a 2-s complement signed integer. This is reverse to *integer2bits*.

Examples:

```
>>> bits2integer(b"\x01\x00\x00\x01\x01")
19
```

construct.lib.**byte2int**(*character*)
Converts b'...' character into (0 through 255) integer.

construct.lib.**bytes**(*countorseq=0*)
Backports bytes() from PY3.

construct.lib.**bytes2bits**(*data*)
Converts between bit and byte representations in b-strings.

Example:

```
>>> bytes2bits(b'ab')
b"\x00\x01\x01\x00\x00\x00\x00\x01\x00\x01\x01\x00\x00\x00\x01\x00"
```

construct.lib.**bytes2integer**(*data*, *signed=False*)
Converts a byte-string into an integer. This is reverse to *integer2bytes*.

Examples:

```
>>> bytes2integer(b'\x00\x00\x00\x13')
19
```

construct.lib.**bytes2integers**(*data*)
Converts bytes into bytes/bytearray, so indexing/iterating yields integers.

construct.lib.**bytes2str**(*string*)
Converts b'...' string into '...' string. On PY2 they are equivalent. On PY3 its utf8 decoded.

construct.lib.**bytestringtype**
alias of __builtin__.str

class construct.lib.**Container**(**args*, ***entrieskw*)
Generic ordered dictionary that allows both key and attribute access, and preserves key order by insertion. Adding keys is preferred using **entrieskw (requires Python 3.6). Equality does NOT check item order. Also provides regex searching.

Example:

```
# empty dict
>>> Container()
# list of pairs, not recommended
>>> Container([ ("name","anonymous"), ("age",21) ])
# This syntax requires Python 3.6
>>> Container(name="anonymous", age=21)
# This syntax is for internal use only
>>> Container(name="anonymous")(age=21)
# copies another dict
>>> Container(dict2)
>>> Container(container2)
```

```
>>> print(repr(obj))
Container(text='utf8 decoded string...')(value=123)
>>> print(obj)
Container
    text = u'utf8 decoded string...' (total 22)
    value = 123
```

clear()
Removes all items.

copy() → a shallow copy of D

items() → list of D's (key, value) pairs, as 2-tuples

keys() → list of D's keys

pop(*key*)
Removes and returns the value for a given key, raises KeyError if not found.

popitem()
Removes and returns the last key and value from order.

search(*pattern*)
Searches a container (non-recursively) using regex.

search_all(*pattern*)
Searches a container (recursively) using regex.

update(*seqordict*)
Appends items from another dict/Container or list-of-tuples.

values() → list of D's values

class construct.lib.**HexDisplayedBytes**
Used internally.

class construct.lib.**HexDisplayedDict**
Used internally.

class construct.lib.**HexDisplayedInteger**
Used internally.

construct.lib.**hexdump**(*data*, *linesize*)
Turns bytes into a unicode string of the format:

```
>>>print(hexdump(b'0' * 100, 16))
hexundump(\"\"\"
0000   30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30   0000000000000000
0010   30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30   0000000000000000
0020   30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30   0000000000000000
0030   30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30   0000000000000000
0040   30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30   0000000000000000
0050   30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30   0000000000000000
0060   30 30 30 30                                       0000
\"\"\")
```

class construct.lib.**HexDumpDisplayedBytes**
Used internally.

class construct.lib.**HexDumpDisplayedDict**
Used internally.

construct.lib.**hexlify**(*data*)
Returns binascii.hexlify(data).

construct.lib.**hexundump**(*data*, *linesize*)
Reverse of *hexdump*.

construct.lib.**int2byte**(*character*)
Converts (0 through 255) integer into b'...' character.

construct.lib.**integer2bits**(*number*, *width*)
Converts an integer into its binary representation in a bit-string. Width is the amount of bits to generate. If width is larger than the actual amount of bits required to represent number in binary, sign-extension is used. If it's smaller, the representation is trimmed to width bits. Each bit is represented as either \x00 or \x01. The most significant is first, big-endian. This is reverse to *bits2integer*.

Examples:

```
>>> integer2bits(19, 8)
b'\x00\x00\x00\x01\x00\x00\x01\x01'
```

construct.lib.**integer2bytes**(*number*, *width*)
Converts a bytes-string into an integer. This is reverse to *bytes2integer*.

Examples:

```
>>> integer2bytes(19,4)
'\x00\x00\x00\x13'
```

construct.lib.**integers2bytes**(*ints*)
Converts integer generator into bytes.

construct.lib.**iteratebytes**(*data*)
Iterates though b'...' string yielding b'...' characters.

construct.lib.**iterateints**(*data*)
Iterates though b'...' string yielding (0 through 255) integers.

class construct.lib.**ListContainer**
Generic container like list. Provides pretty-printing. Also provides regex searching.

Example:

```
>>> ListContainer()
>>> ListContainer([1, 2, 3])
```

```
>>> print(repr(obj))
[1, 2, 3]
>>> print(obj)
ListContainer
    1
    2
    3
```

search(*pattern*)
Searches a container (non-recursively) using regex.

search_all(*pattern*)
Searches a container (recursively) using regex.

construct.lib.**reprstring**(*data*)
Ensures there is b- u- prefix before the string.

construct.lib.**setGlobalPrintFalseFlags**(*enabled=False*)
When enabled, Container __str__ that was produced by FlagsEnum parsing prints all values, otherwise and by default, it prints only the values that are True.

Parameters **enabled** – bool

construct.lib.**setGlobalPrintFullStrings**(*enabled=False*)
When enabled, Container __str__ produces full content of bytes and unicode strings, otherwise and by default, it produces truncated output (16 bytes and 32 characters).

Parameters **enabled** – bool

construct.lib.**setGlobalPrintPrivateEntries**(*enabled=False*)
When enabled, Container __str__ shows keys like _ _index _etc, otherwise and by default, it hides those keys. __repr__ never shows private entries.

Parameters **enabled** – bool

construct.lib.**str2bytes**(*string*)
Converts '...' string into b'...' string. On PY2 they are equivalent. On PY3 its utf8 encoded.

construct.lib.**str2unicode**(*string*)
Converts '...' string into u'...' string. On PY2 its utf8 encoded. On PY3 they are equivalent.

construct.lib.**swapbitsinbytes**(*data*)
Performs a bit-reversal within a byte-string.

Example:

```
>>> swapbits(b'\xf0')
b'\x0f'
```

construct.lib.**swapbytes**(*data*)
Performs an endianness swap on byte-string.

Example:

```
>>> swapbytes(b'abcd')
b'dcba'
```

construct.lib.**swapbytesinbits**(*data*)
Performs an byte-swap within a bit-string. Its length must be multiple of 8.

Example:

```
>>> swapbytesinbits(b'0000000011111111')
b'1111111100000000'
```

construct.lib.**trimstring**(*data*)
Trims b- u- prefix

construct.lib.**unhexlify**(*data*)
Returns binascii.unhexlify(data).

construct.lib.**unicode2str**(*string*)
Converts u'...' string into '...' string. On PY2 its utf8 decoded. On PY3 they are equivalent.

construct.lib.**unicodestringtype**
alias of __builtin__.unicode

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

Index

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V