Construct Documentation

Release 2.8

Arkadiusz Bulski

Aug 22, 2017

Contents

Construct is a powerful **declarative** 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 what you require
- Pointers: jump from here to there in the data stream

CHAPTER 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 a `Range` in that latter is homogenous:

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

See more examples of file formats and network protocols in the repository.

CHAPTER 2

Development and support

Please use the github issues to ask general questions, make feature requests, report issues and bugs, and to send in patches. There is also the mailing list but GitHub should be preffered.

Construct's main documentation is at construct.readthedocs.org, where you can find all kinds of examples. The library itself is developed on github. Releases are also available on pypi.

Construct3 is a different project. It is a rewrite from scratch and belongs to another developer, it diverged from this project. As far as I can tell, it was not released and abandoned.

CHAPTER 3

Requirements

Construct should run on any Python 2.7 3.3 3.4 3.5 3.6 and pypy pypy3 implementation.

Best should be 3.6 because it supports ordered keyword arguments which comes handy when declaring Struct members or manually crafting Containers.

CHAPTER 4

User Guide

Introduction

What is Construct?

In a nutshell, Construct is a declarative binary parser and builder library. To break that down into each different part, Construct is...

Declarative

Construct does not force users to write code in order to create parsers and builders. Instead, Construct gives users a **domain-specific language** for specifying their data structures. Parsing and building procedures are already defined.

Binary

Construct operates on bytes, not strings, and is specialized for binary data. In the past, there was an experimental support for parsing text but it was dropped in version 2.5. That does not mean you cant use fields like unicode CStrings.

Parser and Builder

Structures declared in Construct are symmetrical and describe both the parser and the builder. This eliminates the possibility of disparity between the parsing and building actions, and reduces the amount of code required to implement a format.

Library

Construct is not a framework. It does not have any dependencies besides the Python standard library, and does not require users to adapt their code to its whims.

What is Construct good for?

Construct has been used to parse:

- Networking formats like IP, DNS, SSL
- Binary file formats like Bitmaps, JPEG, PNG
- Executable binaries formats like ELF32, PE32
- Filesystem layouts like Ext2, Fat16, MBR

And many other things! Just look into the examples folder.

What isn't Construct good at?

As previously mentioned, Construct is not a good choice for parsing text, due to the typical complexity of text-based grammars and the relative difficulty of parsing Unicode correctly. In the past, there was an experimental support for parsing text but it was dropped in version 2.5.

Transition to 2.8

Overall

All fields and complex constructs are now nameless. Look at Struct and Sequence, and also Range and Array.

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

FlagsEnum remains

Strings

String remains

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

CString dropped *char_field*

GreedyString dropped *char_field*

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

Structures and Sequences

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

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

On Python 3.6 you can use `Struct(num=Int32ub, text=CString())`

Ranges and Arrays

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

PrefixedArray takes explicit *lengthfield* before subcon

Range uses syntax like `Byte[0:] Byte[:10] Byte[0:10]` and `Range(min=?, max=?, Byte)`

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 now returns a paramterless lambda that returns the parsed object

OnDemandPointer dropped *force_build* parameter

LazyBound remains

Padding and Alignment

Aligned takes explicit *modulus* before the subcon

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

Padding remains

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 or none

Select remains

Miscellaneous and others

Value was made Computed

Embed was made Embedded

Const incorporated Magic field

Pass remains but Terminator was renamed Terminated

Error added

OneOf Noneof remain

Filter added

LengthValueAdapter was replaced by 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 added or remain

SeqOfOne was replaced by FocusedSeq

Numpy added

NamedTuple added

Check added

Default added

Alias was removed

StopIf added

Stream manipulation

Bitwise was reimplemented using Restreamed, and Bytewise was added

Restreamed and Rebuffered were redesigned

Anchor was made Tell and a singleton

Seek was added

Pointer remains

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 added

Checksum and Compressed added

The Basics

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("\x01\x02")
258
>>> Int16ul.parse("\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
```

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 int or raise SizeofError depending on circumstances. Array size depends on whether count of elements is constant (can be a context lambda) and subcon is constant size.

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

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 sequenced collection of fields or other components, that are parsed/built in that order.

```
>>> 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 python 3.6 due to unordered keyword arguments:

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

Operator + can also be used to make Structs, and to merge (embed) them.

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

```
>>> st = ("a"/Byte + "b"/Byte) + "c"/Byte
>>> st.parse(b"abc")
Container(a=97)(b=98)(c=99)
```

Containers

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

```
>>> c = Struct("a"/Byte, "b"/Int16ul, "c"/Single)
>>> x = c.parse(b"\x07\x00\x01\x00\x00\x00\x01")
>>> x
Container(a=7)(b=256)(c=1.401298464324817e-45)
>>> x.b
256
>>> x["b"]
256
>>> print(x)
Container:
    a = 7
    b = 256
    c = 1.401298464324817e-45
```

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 not supported.

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

Building and parsing

And here is how we build Structs and others:

```
>>> # Rebuilding and reparsing from returned...
>>> format = Byte[10]
>>> format.build([1,2,3,4,5,6,7,8,9,0])
b'\x01\x02\x03\x04\x05\x06\x07\x08\t\x00'
>>> format.parse(_)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
>>> format.build(_)
b'\x01\x02\x03\x04\x05\x06\x07\x08\t\x00'
```

```
>>> # Mutate the parsed object and rebuild...
>>> st = Struct("num" / Int32ul)
>>> st.build(dict(num=7890))
b'\xd2\x1e\x00\x00'
```

```
>>> x = st.parse(_)
>>> x
Container(num=7890)
>>> x.num = 555
>>> st.build(x)
b'+\x02\x00\x00'
```

Note: Building is fully duck-typed and can be done with any object.

```
>>> c = Struct("b"/Int32ul, "c"/Flag)
>>> class Dummy:
...     def __getitem__(self, key):
...             return 1
...
>>> dummy = Dummy()
>>> c.build(dummy)
b'\x01\x00\x00\x00\x01'
```

Nesting and embedding

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

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

A Struct can be embedded into an enclosing Struct. This means all the fields of the embedded Struct will be 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, inner fields usually overtake the others.

```
>>> outer = Struct(
...     "data" / Byte,
...     "inner" / Embedded(Struct(
...             "data" / Bytes(4),
...     ))
... )
>>> outer.parse(b"01234")
Container(data=b'1234')
```

```
>>> outer = Struct(
...     "data" / Byte,
...     Embedded(st),
... )
>>>
>>> outer.parse(b"01234")
Container(data=48)(inner=Container(data=b'1234'))
```

As you can see, Containers provide human-readable representations of the data, which is very important for large data structures.

See also:

The *Embedded()* macro.

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.

Operator >> can be used to make Sequences, or to merge them.

Building and parsing

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

Nesting and embedding

Like Structs, Sequences are compatible with the Embedded wrapper. Embedding one Sequence into another causes a merge of the parsed lists of the two Sequences.

```
>>> nseq = Sequence(Byte, Byte, Sequence(Byte, Byte))
>>> nseq.parse(b"abcd")
[97, 98, [99, 100]]
```

```
>>> nseq = Sequence(Byte, Byte, Embedded(Sequence(Byte, Byte)))
>>> nseq.parse(b"abcd")
[97, 98, 99, 100]
```

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 int. Meta-repeaters take values at parse/build time from the context and they will be covered in the meta-constructs tutorial. Ranges differ from Sequences in that they are homogenous, they process elements of same kind. We have four kinds of repeaters. For those of you who wish to look under the hood, two of these repeaters are actually wrappers around Range.

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

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

Ranges are similar but they take a range (pun) of element counts. User can specify the minimum and maximum count.

```
>>> Byte[3:5].parse(b"1234")
[49, 50, 51, 52]
>>> Byte[3:5].parse(b"12")
construct.core.RangeError: expected 3 to 5, found 2
>>> Byte[3:5].build([1,2,3,4,5,6,7])
construct.core.RangeError: expected from 3 to 5 elements, found 7
```

GreedyRange is essentially a Range from 0 to infinity.

```
>>> Byte[:].parse(b"dsadhsaui")
[100, 115, 97, 100, 104, 115, 97, 117, 105]
>>> Byte[:].min
0
>>> Byte[:].max
9223372036854775807
```

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 just the same.

Note that all elements accumulated during parsing are provided as additional lambda parameter.

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

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

The Advanced

Integers and floats

Basic computer science 101. All integers follow Int{8,16,24,32,64}{u,s}{b,l,n} and floats follow Float{32,64}{b,l} a naming pattern. Endianness can be either big-endian, little-endian or native. Integers can be signed or unsigned (non-negative only). Floats do not have a native endianness nor 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/Double among floats.

```
Byte    <-->  Int8ub
Short   <-->  Int16ub
Int     <-->  Int32ub
Long    <-->  Int64ub
Single  <-->  Float32b
Double  <-->  Flaot64b
```

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'
```

Numbers of processor sizes are implemented using *struct* module, others use a custom field.

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

Bytes and bits

"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.

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

Strings

Warning: Strings in Construct work very much like strings in other languages. Be warned however, that Python 2 used byte strings that are now called *bytes*. Python 3 introduced unicode strings which require an encoding to be used, utf-8 being the best option. When no encoding is provided on Python 3, those constructs work on byte strings similar to Bytes and GreedyBytes fields. Encoding can be set once, globally using `setglobalstringencoding()` or provided with each field separately.

String is a fixed-length construct that pads builded string with null bytes, and strips those same null bytes when parsing. Note that some encodings do not work properly because they return null bytes within the encoded stream, utf-16 and utf-32 for example.

```
>>> String(10).build(b"hello")
b'hello\x00\x00\x00\x00\x00'
```

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

You can use different bytes for padding (although they will break any encoding using those within the stream). Strings can also be trimmed when building. If you supply a too long string, the construct will chop if apart instead of raising a StringError.

```
>>> String(10, padchar=b"XYZ", paddir="center").build(b"abc")
b'XXXabcXXXX'
```

```
>>> String(10, trimdir="right").build(b"12345678901234567890")
b'1234567890'
```

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 0 byte at the end. Note that the length field can be variable length itself, as shown below. VarInt should be preferred when building new protocols.s

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

CString is an another variable length string, 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.

Warning: Do not use >1 byte encodings like UTF16 or UTF32 with CStrings.

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

Last but not least, a GreedyString does the same thing that GreedyBytes does. It reads until the end of stream and decodes it using the specified encoding.

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

Other short fields

Booleans are flags:

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

Enums translate between string names and usually integer vaues:

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

```
>>> FlagsEnum(Byte, a=1, b=2, c=4, d=8).parse(b"\x03")
Container(c=False)(b=True)(a=True)(d=False)
```

The Bit/Byte Duality

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 Bit Octet BitsInteger handle parsing and building bit strings. There are also few fields like Struct and Flag that work with both byte-strings and bit-strings.

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 1's and 0's), 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:

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

Note: BitStruct is actually just a wrapper for the `Bitwise()` around a `Struct()` construct.

Important notes

- BitStructs are non-nestable so writing something like `BitStruct("foo", BitStruct("bar", Octet("spam")))` will not work. You can use regular Structs inside BitStructs.
- BitStructs are embeddable. The `Embedded` wrapper can be used for that purpose. There is also the `EmbeddedBitStruct`.
- 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 of the amount of bits and bytes does not align properly.
- Pointers and OnDemand - Do not place Pointers or OnDemands inside bitwise because it uses an internal stream, so external stream offsets will turn out wrong, have side-effects or raise exceptions.
- Advanced classes like tunneling may not work in bitwise context. Only basic fields like integers were throughly tested.

Integers out of bits

`construct.`**`BitsInteger`**(*length*, *signed=False*, *swapped=False*, *bytesize=8*)

A byte field, that parses into and builds from integers as opposed to b-strings. This is similar to Bit/Nibble/Octet fields but can have arbitrary sizes. This must be enclosed in Bitwise.

Parameters

- **`length`** – number of bits in the field, or a context function that returns int
- **`signed`** – whether the value is signed (two's complement), default is False (unsigned)
- **`swapped`** – whether to swap byte order (little endian), default is False (big endian)
- **`bytesize`** – size of byte as used for byte swapping (if swapped), default is 8

Example:

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

Convenience wrappers for BitsInteger

Bit A single bit

Nibble A sequence of 4 bits (half a byte)

Octet An sequence of 8 bits (byte)

Fields that do both

Most simple fields (such as Flag, Padding, Terminated, etc.) are ignorant to the granularity of the data they operate on. The actual granularity depends on the enclosing layers.

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

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

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

```
>>> format = BitStruct(
...     Padding(2),
...     "x" / Flag,
...     Padding(5),
... )
>>> format.build(dict(x=5))
b' '
```

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.

Adapters and Validators

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 of the abstract Adapter class, and implement their own versions of `_encode` and `_decode`, as shown below:

```
>>> class IpAddressAdapter(Adapter):
...     def _encode(self, obj, context):
...         return list(map(int, obj.split(".")))
...     def _decode(self, obj, context):
...         return "{0}.{1}.{2}.{3}".format(*obj)
...
>>> IpAddress = IpAddressAdapter(Byte[4])
```

As you can see, the IpAddressAdapter encodes strings of the format "XXX.XXX.XXX.XXX" to a binary string of 4 bytes, and decodes such binary strings into the more readable "XXX.XXX.XXX.XXX" format. Also note that the adapter does not perform any manipulation of the stream, it only converts between the objects!

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 we'll see with enums in a moment, we can use the same enum for `Byte` or `Int32sl` or `Float64l`, as long as the underlying construct returns an object we can map. Moreover, we can stack several adapters on top of one another, to create a nested adapter.

Enums

Enums provide symmetrical name-to-value mapping. The name may be misleading, as it's not an enumeration as you would expect in C. But since enums in C are often just used as a collection of named values, we'll stick with the name. Hint: enums are implemented by the `Mapping` adapter, which provides mapping of values to other values (not necessarily names to numbers).

```
>>> c = Enum(Byte, TCP=6, UDP=17)
>>> c.parse(b"\x06")
'TCP'
>>> c.build("UDP")
b'\x11'
>>> c.build(17)
b'\x11'
```

We can also supply a default mapped value when no mapping exists for them. We do this by supplying a keyword argument named `default`. If we don't supply a default value, an exception is raised.

```
>>> c = Enum(Byte, TCP=6, UDP=17)
>>> c.parse(b"\xff")
construct.core.MappingError: no decoding mapping for 255
>>> c.build("unknown")
construct.core.MappingError: no encoding mapping for 'unknown'
```

```
>>> c = Enum(Byte, TCP=6, UDP=17, default=0)
>>> c.parse(b"\xff")
0
>>> c.build(99)
b'\x00'
```

We can also just "pass through" unmapped values. We do this by supplying `default = Pass`. If you are curious, `Pass` is a special construct that "does nothing"; in this context, we use it to indicate the Enum to "pass through" the unmapped value as-is.

```
>>> c = Enum(Byte, TCP=6, UDP=17, default=Pass)
>>> c.parse(b"\xff")
255
```

Using expressions instead of classes

Adaters can be created declaratively using ExprAdapter:

```
>>> IpAddress = ExprAdapter(Byte[4],
...     encoder = lambda obj,ctx: list(map(int, obj.split("."))),
...     decoder = lambda obj,ctx: "{0}.{1}.{2}.{3}".format(*obj), )
```

Validating

Validating means making sure the parsed/built object meets a given condition. Validators simply raise the `ValidatorError` if the object is invalid. They are usually used to make sure a "magic number" is found, the correct version of the protocol, a file signature is matched. You can write custom validators by deriving from the Validator class and implementing the `_validate` method. This allows you to write validators for more complex things, such as making sure a CRC field (or even a cryptographic hash) is correct.

The two most common cases already exist as builtins.

class `construct.`**`NoneOf`**

Validates that the object is none of the listed values, both during parsing and building.

Parameters

- **`subcon`** – a construct to validate
- **`invalids`** – a collection implementing *in*

See also:

Look at *`OneOf()`* for examples, works the same.

class `construct.`**`OneOf`**

Validates that the object is one of the listed values, both during parsing and building.

Parameters

- **`subcon`** – a construct to validate
- **`valids`** – a collection implementing *in*

Example:

```
>>> OneOf(Byte, [1,2,3]).parse(b"\x01")
1
>>> OneOf(Byte, [1,2,3]).parse(b"\x08")
construct.core.ValidationError: ('invalid object', 8)

>>> OneOf(Bytes(1), b"1234567890").parse(b"4")
b'4'
>>> OneOf(Bytes(1), b"1234567890").parse(b"?")
construct.core.ValidationError: ('invalid object', b'?')

>>> OneOf(Bytes(2), b"1234567890").parse(b"78")
b'78'
>>> OneOf(Bytes(2), b"1234567890").parse(b"19")
construct.core.ValidationError: ('invalid object', b'19')
```

Notice that *OneOf(dtype, [value])* is essentially equivalent to *Const(dtype, value)*.

class `construct.`**`Filter`**

Filters a list leaving only the elements that passed through the validator.

Parameters

- **`subcon`** – a construct to validate, usually a Range or Array or Sequence
- **`predicate`** – a function taking (obj, context) and returning a bool

Example:

```
>>> Filter(obj_ != 0, Byte[:]).parse(b"\x00\x02\x00")
[2]
>>> Filter(obj_ != 0, Byte[:]).build([0,1,0,2,0])
b'\x01\x02'
```

Using expressions instead of classes

Validators can be created declaratively using ExprValidator:

```
>>> OneOf = ExprValidator(Byte,
...     validator = lambda obj,ctx: obj in [1,3,5])
```

Checking

Checks can also be made using the context, being done just in the middle of parsing or building and not on a particular object.

class `construct.`**`Check`**(*func*)

Checks for a condition, and raises ValidationError if the check fails.

Example:

```
Check(lambda ctx: len(ctx.payload.data) == ctx.payload_len)

Check(len_(this.payload.data) == this.payload_len)
```

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 (parsing or building). In other words, meta constructs are self-referring. The context is a dictionary that is created during the construction process by Structs and Sequences, and is “propagated” down and up to all constructs along the way, so that they could use it. 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:

```
>>> class PrintContext(Construct):
...     def _parse(self, stream, context, path):
...         print(context)
...
>>> st = Struct(
...     "a" / Byte,
...     PrintContext(),
...     "b" / Byte,
...     PrintContext(),
... )
>>> 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" 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(lambda ctx: ctx["count"]),
... )
>>> st.parse(b"\x05abcde")
Container(count=5)(data=b'abcde')
```

Of course the function can return anything (it doesn't have to use ctx at all):

```
>>> st = Struct(
...     "ct" / Computed(lambda ctx: 7),
... )
>>> st.parse(b"")
Container(ct=7)
```

And here's how we use the special '_' name to get to the upper layer. Here the length of the string is calculated as `length1 + length2`:

```
>>> st = Struct(
...     "length1" / Byte,
...     "inner" / Struct(
...             "length2" / Byte,
...             "sum" / Computed(lambda ctx: ctx._.length1 + ctx.length2),
...     ),
... )
>>> st.parse(b"12")
Container(length1=49)(inner=Container(length2=50)(sum=99))
```

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 even a better syntax. The *this* singleton object can be used to build a lambda expression. All four examples below are equivalent:

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

Of course, *this* can be mixed with other calculations. When evaluating, each instance of this is replaced by ctx.

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

Using *len_* builtin alikes

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

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

These can be used in newly added Rebuild wrappers that take compute count/length fields from another items-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'
```

Incidentally, when the count field is directly before the items field you can also use PrefixedArray. However in some protocols these fields are separate and the other approach is good:

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

Using *obj_* expression

There is also an analog that takes both (obj, context) unlike the *this* singleton which only takes a context:

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

These can be used in few classes that use (obj, context) lambdas:

```
>>> RepeatUntil(obj_ == 0, Byte).build([1,2,0,1,0])
b'\x01\x02\x00'
```

Array

When creating an Array, rather than specifying a constant length, you can instead specify that it repeats a variable number of times.

```
PrefixedArray  <-->  FocusedSeq(1,
    "count"/Rebuild(lengthfield, len_(this.items)),
    "items"/subcon[this.count],
```

RepeatUntil

A repeater that repeats until a condition is met. The perfect example is null-terminated strings.

Note: For null-terminated strings, use *CString()*.

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

Switch

Branches the construction path based on a condition, similarly to C's switch statement.

```
>>> 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')
```

When the condition is not found in the switching table, and a default construct is not given, an exception is raised (SwitchError). In order to specify a default construct, set default (a keyword argument) when creating the Switch. Note that default is a construct, not a value.

```
>>> st = Struct(
...     "type" / Byte,
...     "data" / Switch(this.type, {
...             1 : Int8ul,
...             2 : Int8sl,
...         }, default = Int8ul),
... )
>>> st.parse(b"\xff\x01")
Container(type=255)(data=1)
```

When you want to ignore/skip errors, you can use the Pass construct, which is a no-op construct. Pass will simply return None, without reading anything from the stream. Pass will also not put anything into the stream.

```
>>> st = Struct(
...     "type" / Byte,
...     "data" / Switch(this.type, {
...             1 : Int8ul,
...             2 : Int8sl,
...     }, default = Pass),
... )
>>> st.parse(b"??????")
Container(type=63)(data=None)
```

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 ctx: not ctx.flag1 or ctx.flag2 and ctx.flag3
```

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

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

Miscellaneous

Miscellaneous

Embedded

Embeds a struct into the enclosing struct, merging fields. Can also embed sequences into sequences.

```
>>> Struct("a"/Byte, Embedded(Struct("b"/Byte)), "c"/Byte).parse(b"abc")
Container(a=97)(b=98)(c=99)
```

Const

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

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

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

```
>>> Const(Int32ul, 1).build(None)
b'\x01\x00\x00\x00'
```

Computed

Represents a computed value. Value does not read or write anything to the stream. It only returns its computed value as the result. Usually Computed fields are used for computations on the context. Look at the 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'
```

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

Pass

A do-nothing construct, useful in Switches and Enums.

Note: Pass is a singleton object. Do not try to instantiate it, `Pass()` will not work.

```
>>> Pass.parse(b"123123")
>>> Pass.build(None)
b''
```

Terminated

Asserts the end of the stream has been reached at the point it was placed. You can use this to ensure no more unparsed data follows.

Note: Terminated is a singleton object. Do not try to instantiate it, `Terminated()` will not work.

```
>>> Terminated.parse(b"")
>>> Terminated.parse(b"x")
construct.core.TerminatedError: expected end of stream
```

Numpy

Numpy arrays can be preserved and retrived along with their dtype, shape and size, and all. Otherwise, if dtype is constant, you could use PrefixedArray or Range to store enumerables.

```
>>> import numpy
>>> Numpy.build(numpy.asarray([1,2,3]))
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.

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

Rebuild

When there is an array separated from its length field, the Rebuild wrapped can be used to measure the len of the list at building. Note that both the *len_* and *this* expressions are used as discussed in meta chapter.

```
>>> 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'
```

Check

When fields are expected to be coherent in some way but integrity cannot be checked easily using Const or otherwise, then a Check field can be put in place that will compute from the context if the integrity is preserved. For example, maybe there is a count field (implied being non-negative but the field is signed):

```
>>> 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])
FieldError: packer '>B' error during building, given value 9090
...
>>> st = Struct(integrity=Check(this.count == len_(this.items)), count=Byte,␣
↪items=Byte[this.count])
ValidationError: check failed during building
```

FocusedSeq

When a sequence is has some fields that could be ommited like Const and Terminated, user can focus on the particular fields that are useful:

```
>>> d = FocusedSeq("num", Const(b"MZ"), "num"/Byte, Terminated)
>>> d = FocusedSeq(1,     Const(b"MZ"), "num"/Byte, Terminated)
...
```

```
>>> d.parse(b"MZ\xff")
255
>>> d.build(255)
b'MZ\xff'
```

Default

Allows to make a field have a default value, which comes handly when building a Struct from a dict with missing keys.

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

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 if None, 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 the *parsefrom* parameter then stream will be left back at the starting offset. Many users fail to use this class properly.

```
>>> Union(0, "raw"/Bytes(8), "ints"/Int32ub[2], "shorts"/Int16ub[4], "chars"/Byte[8]).
↪parse(b"12345678")
Container(raw=b'12345678')(ints=[825373492, 892745528])(shorts=[12594, 13108, 13622,␣
↪14136])(chars=[49, 50, 51, 52, 53, 54, 55, 56])
```

```
>>> Union(0, "raw"/Bytes(8), "ints"/Int32ub[2], "shorts"/Int16ub[4], "chars"/Byte[8]).
↪build(dict(chars=range(8)))
b'\x00\x01\x02\x03\x04\x05\x06\x07'
```

```
Note that this syntax works ONLY on python 3.6 due to unordered keyword arguments:
>>> Union(0, raw=Bytes(8), ints=Int32ub[2], shorts=Int16ub[4], chars=Byte[8])
>>> 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.

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

```
Note that this syntax works ONLY on python 3.6 due to unordered keyword arguments:
>>> Select(num=Int32ub, text=CString(encoding="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.

```
>>> Optional(Int64ul).parse(b"1234")
>>> Optional(Int64ul).parse(b"12345678")
4050765991979987505
```

```
>>> Optional(Int64ul).build(1)
b'\x01\x00\x00\x00\x00\x00\x00\x00'
>>> Optional(Int64ul).build("1")
b''
```

If

Parses or builds the subconstruct only if a certain condition is met. Otherwise, returns a None and puts nothing.

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

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

Switch

Branches the construction based on a return value from a function. This is a more general version of IfThenElse.

Warning: You can use Embedded(Switch(...)) but not Switch(Embedded(...)). Sames applies to If and IfThenElse macros.

```
>>> Switch(this.n, { 1:Byte, 2:Int32ub }).build(5, dict(n=1))
b'\x05'
>>> Switch(this.n, { 1:Byte, 2:Int32ub }).build(5, dict(n=2))
b'\x00\x00\x00\x05'
```

StopIf

Checks for a condition, and stops a Struct Sequence Range from parsing or building.

Warning: May break sizeof methods. Unsure.

```
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)))
```

Alignment and Padding

Padding

Adds and removes bytes without returning the to the user. Analog to Padded but does not wrap around another construct.

```
>>> Padding(4).build(None)
b'\x00\x00\x00\x00'
>>> Padding(4, strict=True).parse(b"****")
construct.core.PaddingError: expected b'\x00\x00\x00\x00', found b'****'
```

Padded

Appends additional null bytes to achieve a fixed length.

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

Aligned

Aligns the subconstruct to a given modulus boundary.

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

AlignedStruct

Automatically aligns all the fields of the Struct to the modulus boundary. It does NOT align entire Struct.

```
>>> AlignedStruct(4, "a"/Int8ub, "b"/Int16ub).build(dict(a=1,b=5))
b'\x01\x00\x00\x00\x00\x05\x00\x00'
```

Streaming tactics

Note: Certain constructs are available only for seekable streams (in-memory and files) and some require tellable streams (which in fact is a subset of seekability). Sockets and pipes do not support seeking, so you'll have to first read the data from the stream, and parse it in-memory, or use the `Rebuffered()` wrapper.

Wrappers

Pointer

Pointer allows for non-sequential construction. The pointer first changes the stream position, does the construction, and restores the original stream position.

```
>>> Pointer(8, Bytes(1)).parse(b"abcdefghijkl")
b'i'
>>> Pointer(8, Bytes(1)).build(b"x")
b'\x00\x00\x00\x00\x00\x00\x00\x00x'
```

Peek

Parses the subconstruct but restores the stream position afterwards (it does peeking).

```
>>> Sequence(Peek(Byte), Peek(Int16ub)).parse(b"\x01\x02")
[1, 258]
```

Pure side effects

Seek makes a jump within the stream and leaves it at that point. It does not read or write anything to the stream by itself.

class `construct.core.`**`Seek`**(*at*, *whence=0*)

Sets a new stream position when parsing or building. Seeks are useful when many other fields follow the jump. Pointer works when there is only one field to look at, but when there is more to be done, Seek may come useful.

See also:

Analog `Pointer()` wrapper that has same side effect but also processed a subcon.

Parameters

- **`at`** – where to jump to, can be an int or a context lambda
- **`whence`** – is the offset from beginning (0) or from current position (1) or from ending (2), can be an int or a context lambda, default is 0

Example:

```
>>> (Seek(5) >> Byte).parse(b"01234x")
[5, 120]
>>> (Bytes(10) >> Seek(5) >> Byte).build([b"0123456789", None, 255])
b'01234\xff6789'
```

Tell checks the current stream position and returns it, also putting it into the context. It does not read or write anything to the stream by itself.

`construct.core.`**`Tell`**`()`

Gets the stream position when parsing or building.

Tells are 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.

See also:

Better use *`RawCopy()`* instead of manually extracting bytes.

Example:

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

Stream manipulation

`construct.core.`**`Rebuffered`**`(`*subcon*, *tailcutoff=None*`)`

Caches bytes from the underlying stream, so it becomes seekable and tellable. Also makes the stream blocking, in case it came from a socket or a pipe. Optionally, stream can forget bytes that went a certain amount of bytes beyond the current offset, allowing only a limited seeking capability while allowing to process an endless stream.

Warning: Experimental implementation. May not be mature enough.

Parameters

- **`subcon`** – the subcon which will operate on the buffered stream
- **`tailcutoff`** – optional, amount of bytes kept in buffer, by default buffers everything

Example:

```
Rebuffered(RepeatUntil(lambda obj,ctx: ?,Byte), tailcutoff=1024).parse_
↪stream(endless_nonblocking_stream)
```

`construct.core.`**`Restreamed`**`(`*subcon*, *encoder*, *encoderunit*, *decoder*, *decoderunit*, *decoderunitname*, *sizecomputer*`)`

Transforms bytes between the underlying stream and the subcon.

When the parsing or building is done, the wrapper stream is closed. If read buffer or write buffer is not empty, error is raised.

See also:

Both *`Bitwise()`* and *`Bytewise()`* are implemented using Restreamed.

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 after parsing/building. Also do NOT use pointers inside.

Parameters

- **subcon** – the subcon which will operate on the buffer
- **encoder** – a function that takes a b-string and returns a b-string (used when building)
- **encoderunit** – ratio as int, encoder takes that many bytes at once
- **decoder** – a function that takes a b-string and returns a b-string (used when parsing)
- **decoderunit** – ratio as int, decoder takes that many bytes at once
- **decoderunitname** – English string that describes the units (plural) returned by the decoder. Used for error messages.
- **sizecomputer** – a function that computes amount of bytes outputed by some bytes

Example:

```
Bitwise  <--> Restreamed(subcon, bits2bytes, 8, bytes2bits, 1, lambda n: n//8)
Bytewise <--> Restreamed(subcon, bytes2bits, 1, bits2bytes, 8, lambda n: n*8)
```

Tunneling tactics

Obtaining raw bytes

When some value needs to be processed as both a parsed object and raw bytes, both of these can be obtained using RawCopy. You can build from either the object or raw bytes as well.

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

Endianness

When little endianness is needed, either use fields like `Int*l` or swap bytes of an arbitrary field:

```
Int24ul <--> ByteSwapped(Int24ub)
```

```
>>> ByteSwapped(Int32ub).build(0x01020304)
'\x04\x03\x02\x01'
```

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

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

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 an element count (see PrefixedArray) or by prepending a byte count:

```
>>> Prefixed(VarInt, GreedyBytes).parse(b"\x05hello?????")
b'hello'
...
>>>> Prefixed(VarInt, Byte[:]).parse(b"\x03\x01\x02\x03?????")
[1, 2, 3]
```

Note that VarInt encoding should be preferred because it is both compact and never overflows.

Optionally, length field can include its own size. Then the length field must be of fixed size.

Compression and checksuming

Data can be checksummed easily. 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(
        "a" / Byte,
        "b" / Byte,
    )),
    "checksum" / Checksum(Bytes(64), lambda data: hashlib.sha512(data).digest(), this.
↪fields.data),
)
data = d.build(dict(fields=dict(value=dict(a=1,b=2))))
# returned b'\x01\x02\xbd\xd8\x1a\xb23\xbc\xebj\xd23\xcd\x18qP\x93␣
↪\xa1\x8d\x035\xa8\x91\xcf\x98s\t\x90\xe8\x92>\x1d\xda\x04\xf35\x8e\x9c~
↪\x1c=\x16\xb1o@\x8c\xfa\xfbj\xf52T\xef0#\xed$6S8\x08\xb6\xca\x993'
```

Also can be compressed easily. 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.

```
Compressed(GreedyBytes, "zlib")
...
Prefixed(VarInt, Compressed(GreedyBytes, "zlib"))
Struct("inner"/above)
...
Compressed(Struct(...), "zlib")
```

Lazy parsing

Note: Certain constructs are available only for seekable streams (in-memory and files) and some require tellable streams (which in fact is a subset of seekability). Sockets and pipes do not support seeking, so you'll have to first read the data from the stream, and parse it in-memory, or use the *Rebuffered()* wrapper.

Structs Sequences and Ranges

Lazy* constructs allow lazy construction and deconstruction, meaning the data is actually parsed only when it's requested (demanded). Lazy parsing (also called on-demand parsing but that term is reserved here) is very useful with record-oriented data, where you don't have to actually parse the data unless it's actually needed. The result of parsing is a different container that remembers names of the members and their location in the stream, and when the data is accessed by key (or attribute for that matter) then that field is parsed. Members are parsed only once each.

Essentially almost every code that uses the base classes also works on these but there are few things that one has to be aware of when using lazy equivalents.

LazyStruct works like Struct but parses into a LazyContainer.

> Equivalent to Struct construct, however fixed size members are parsed on demand, others are parsed immediately. If entire struct is fixed size then entire parse is essentially one seek.

LazySequence works like Sequence but parses into a LazySequenceContainer.

> Equivalent to Sequence construct, however fixed size members are parsed on demand, others are parsed immediately. If entire sequence is fixed size then entire parse is essentially one seek.

LazyRange works like Range but parses into a LazyRangeContainer.

> Equivalent to Range construct, but members are parsed on demand. Works only with fixed size subcon.

OnDemand

There is a different approach to lazy parsing, where only one field is made lazy. Parsing returns a parameterless lambda that when called, returns the parsed data. Right now, each time the lambda is called the object is parsed again, so it the inner subcon is non-deterministic, each parsing may return a different object. Builds from a parsed object or a lambda.

```
>>> OnDemand(Byte).parse(b"\xff")
<function OnDemand._parse.<locals>.<lambda> at 0x7fdc241cfc80>
>>> _()
255
>>> OnDemand(Byte).build(16)
b'\x10'
```

There is also OnDemandPointer class.

```
>>> OnDemandPointer(lambda ctx: 2, Byte).parse(b"\x01\x02\x03garbage")
<function OnDemand._parse.<locals>.effectuate at 0x7f6f011ad510>
>>> _()
3
```

LazyBound

A lazy-bound construct that binds to the construct only at runtime. Useful for recursive data structures (like linked lists or trees), where a construct needs to refer to itself (while it doesn't exist yet).

```
>>> st = Struct(
...     "value"/Byte,
...     "next"/If(this.value > 0, LazyBound(lambda ctx: st)),
... )
...
>>> st.parse(b"\x05\x09\x00")
Container(value=5)(next=Container(value=9)(next=Container(value=0)(next=None)))
```

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

Extending Construct

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 `_encode` and `_decode`:

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

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

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 more simpler to implement.
- To make things faster, try using pypy, or write your code in cython. The python-level classes are as fast as it gets, assuming generality.

The only reason you might want to write a construct 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 existing construct, you can post a question or request as an Issue, or copy paste its code and modify it.

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

Raw constructs

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

```
class MyConstruct(Construct):
    def _parse(self, stream, context, path):
        # read from the stream (usually not directly)
        # return object
        pass

    def _build(self, obj, stream, context, path):
        # write obj to the stream (usually not directly)
        # no return value is necessary
        pass

    def _sizeof(self, context, path):
        # return computed size (when fixed size or depends on context)
        # or raise SizeofError if not possible (when variable size)
        pass
```

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

Subconstructs

Deriving of class Subconstruct, these wrap an inner construct, inheriting it's properties (name and flags). In their `_parse` and `_build` methods, they will call `self.subcon._parse` or `self.subcon._build` respectively. Most subconstructs do not need to override `_sizeof`.

```
class MySubconstruct(Subconstruct):
    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)
        # no return value is necessary
        # but if returns one, it will be replace previous context value

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

Debugging Construct

Programming data structures in Construct is much easier than writing the equivalent procedural code, both in terms of RAD 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.

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.

```
>>> Struct("count"/Byte, "items"/Byte[this.count], Probe()).parse(b"\x05abcde")
================================================================================
Probe <unnamed 3>
path is parsing, func is None
EOF reached
Container:
    count = 5
    items = ListContainer:
        97
        98
        99
        100
        101
================================================================================
Container(count=5)(items=[97, 98, 99, 100, 101])

>>> (Byte >> Probe()).parse(b"?")
================================================================================
Probe <unnamed 1>
path is parsing, func is None
EOF reached
Container:
    0 = 63
================================================================================
[63, None]
```

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

```
>>> st = "junk"/RepeatUntil(obj_ == 0,Byte) + "num"/Byte + Probe()
>>> st.parse(b"xcnzxmbjskahuiwerhquiehnsdjk\x00\xff")
================================================================================
Probe <unnamed 5>
path is parsing, func is None
EOF reached
Container:
    junk = ListContainer:
        120
        99
        110
        122
        120
        109
        98
        106
        115
        107
        97
        104
        117
        105
        119
        101
        114
        104
        113
        117
        105
        101
        104
        110
        115
        100
        106
        107
        0
    num = 255
================================================================================
Container(junk=[120, 99, 110, 122, 120, 109, 98, 106, 115, 107, 97, 104, 117, 105,␣
↪119, 101, 114, 104, 113, 117, 105, 101, 104, 110, 115, 100, 106, 107, 0])(num=255)
```

```
>>> st = "junk"/RepeatUntil(obj_ == 0,Byte) + "num"/Byte + ProbeInto(this.num)
>>> st.parse(b"xcnzxmbjskahuiwerhquiehnsdjk\x00\xff")
================================================================================
Probe <unnamed 6>
path is parsing, func is this.num
EOF reached
255
================================================================================
Container(junk=[120, 99, 110, 122, 120, 109, 98, 106, 115, 107, 97, 104, 117, 105,␣
↪119, 101, 114, 104, 113, 117, 105, 101, 104, 110, 115, 100, 106, 107, 0])(num=255)
```

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 <Range: None>:
  File "/home/arkadiusz/Dokumenty/GitHub/construct/construct/debug.py", line 116, in _
↪build
    obj.stack.append(a)
  File "/home/arkadiusz/Dokumenty/GitHub/construct/construct/core.py", line 1069, in _
↪build
    raise RangeError("expected from %d to %d elements, found %d" % (self.min, self.
↪max, len(obj)))
construct.core.RangeError: expected from 3 to 3 elements, found 0

> /home/arkadiusz/Dokumenty/GitHub/construct/construct/core.py(1069)_build()
-> raise RangeError("expected from %d to %d elements, found %d" % (self.min, self.max,
↪ len(obj)))
```

```
(Pdb)
================================================================================

>>> format = Struct(
...     "spam" / Debugger(Enum(Byte, A=1,B=2,C=3)),
... )
>>> format.parse(b"\xff")
================================================================================
Debugging exception of <Mapping: None>:
  File "/home/arkadiusz/Dokumenty/GitHub/construct/construct/core.py", line 2578, in _
↪decode
    return self.decoding[obj]
KeyError: 255

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "/home/arkadiusz/Dokumenty/GitHub/construct/construct/debug.py", line 127, in _
↪parse
    return self.subcon._parse(stream, context)
  File "/home/arkadiusz/Dokumenty/GitHub/construct/construct/core.py", line 308, in _
↪parse
    return self._decode(self.subcon._parse(stream, context), context)
  File "/home/arkadiusz/Dokumenty/GitHub/construct/construct/core.py", line 2583, in _
↪decode
    raise MappingError("no decoding mapping for %r" % (obj,))
construct.core.MappingError: no decoding mapping for 255

(you can set the value of 'self.retval', which will be returned)
> /home/arkadiusz/Dokumenty/GitHub/construct/construct/core.py(2583)_decode()
-> raise MappingError("no decoding mapping for %r" % (obj,))
(Pdb) self.retval = "???"
(Pdb) q
```

Error

Raises an exception when triggered by parse or build. Can be used as a sentinel that blows a whistle when a conditional branch goes the wrong way, or to raise an error explicitly the declarative way.

```
>>> d = "x"/Int8sb >> IfThenElse(this.x > 0, Int8sb, Error)
>>> d.parse(b"\xff\x05")
construct.core.ExplicitError: Error field was activated during parsing
```

CHAPTER 5

API Reference

Core API: Bytes and bits

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

A field consisting of a specified number of bytes. Builds from a b-string, or an integer (although deprecated and BytesInteger should be used).

See also:

Analog *BytesInteger()* that parses and builds from integers.

Parameters **length** – an int or a function that takes context and returns int

Example:

```
>>> Bytes(4).parse(b"beef")
b'beef'
>>> Bytes(4).build(_)
b'beef'
>>> Bytes(4).build(255)
b'\x00\x00\x00\xff'
>>> Bytes(4).sizeof()
4
```

construct.**GreedyBytes**()

A byte field, that parses the stream to the end and builds into the stream as-is.

This is an analog to *Bytes(infinity)*, pun intended.

See also:

Analog *GreedyString()* that parses and builds from strings using an encoding.

Example:

```
>>> GreedyBytes.parse(b"helloworld")
b'helloworld'
>>> GreedyBytes.build(b"asis")
b'asis'
```

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

Converts the stream from bytes to bits, and passes the bitstream to underlying subcon.

See also:

Analog *Bytewise()* that transforms subset of bits back to bytes.

Warning: Do not use pointers inside.

Parameters **subcon** – any field that works with bits like: BitStruct BitsNumber Bit Nibble Octet

Example:

```
>>> Bitwise(Octet).parse(b"\xff")
255
>>> Bitwise(Octet).build(1)
b'\x01'
>>> Bitwise(Octet).sizeof()
1
```

construct.**BytesInteger**(*length*, *signed=False*, *swapped=False*, *bytesize=1*)

A byte field, that parses into and builds from integers as opposed to b-strings. This is similar to Int* fields but can have arbitrary size.

See also:

Analog *BitsInteger()* that operates on bits.

Parameters

- **length** – number of bytes in the field, or a function that takes context and returns int
- **signed** – whether the value is signed (two's complement), default is False (unsigned)
- **swapped** – whether to swap byte order (little endian), default is False (big endian)
- **bytesize** – size of byte as used for byte swapping (if swapped), default is 1

Example:

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

construct.**BitsInteger**(*length*, *signed=False*, *swapped=False*, *bytesize=8*)

A byte field, that parses into and builds from integers as opposed to b-strings. This is similar to Bit/Nibble/Octet fields but can have arbitrary sizes. This must be enclosed in Bitwise.

Parameters

- **length** – number of bits in the field, or a context function that returns int

- **signed** – whether the value is signed (two's complement), default is False (unsigned)
- **swapped** – whether to swap byte order (little endian), default is False (big endian)
- **bytesize** – size of byte as used for byte swapping (if swapped), default is 8

Example:

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

Core API: Strings

`construct.`**`setglobalstringencoding`**`(`*encoding*`)`

Sets the encoding globally for all String PascalString CString GreedyString instances.

Parameters **encoding** – a string like "utf8" etc or None, which means working with bytes

`construct.`**`String`**`(`*length*, *encoding=None*, *padchar='\x00'*, *paddir='right'*, *trimdir='right'*`)`

A configurable, fixed-length or variable-length string field.

When parsing, the byte string is stripped of pad character (as specified) from the direction (as specified) then decoded (as specified). Length is a constant integer or a function of the context. When building, the string is encoded (as specified) then padded (as specified) from the direction (as specified) or trimmed as bytes (as specified).

The padding character and direction must be specified for padding to work. The trim direction must be specified for trimming to work.

Parameters

- **length** – length in bytes (not unicode characters), as int or context function
- **encoding** – encoding (e.g. "utf8") or None for bytes
- **padchar** – b-string character to pad out strings (by default b"x00")
- **paddir** – direction to pad out strings (one of: right left both)
- **trimdir** – direction to trim strings (one of: right left)

Example:

```
>>> String(10).build(b"hello")
b'hello\x00\x00\x00\x00\x00'
>>> String(10).parse(_)
b'hello'
>>> String(10).sizeof()
10

>>> String(10, encoding="utf8").build("")
b'\xd0\x90\xd1\x84\xd0\xbe\xd0\xbd\x00\x00'
>>> String(10, encoding="utf8").parse(_)
''

>>> String(10, padchar=b"XYZ", paddir="center").build(b"abc")
b'XXXabcXXXX'
```

```
>>> String(10, padchar=b"XYZ", paddir="center").parse(b"XYZabcXYZY")
b'abc'

>>> String(10, trimdir="right").build(b"12345678901234567890")
b'1234567890'
```

`construct.`**`PascalString`**(*lengthfield*, *encoding=None*)

A length-prefixed string.

`PascalString` is named after the string types of Pascal, which are length-prefixed. Lisp strings also follow this convention.

The length field will not appear in the same dict, when parsing. Only the string will be returned. When building, actual length is prepended before the encoded string. The length field can be variable length (such as VarInt). Stored length is in bytes, not characters.

Parameters

- **`lengthfield`** – a field used to parse and build the length
- **`encoding`** – encoding (eg. "utf8") or None for bytes

Example:

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

`construct.`**`CString`**(*terminators='\x00'*, *encoding=None*)

A string ending in a terminator b-string character.

`CString` is similar to the strings of C.

By default, the terminator is the NULL byte (b'x00'). Terminators field can be a longer b-string, and any of the characters breaks parsing. First terminator byte is used when building.

Parameters

- **`terminators`** – sequence of valid terminators, first is used when building, all are used when parsing
- **`encoding`** – encoding (e.g. "utf8") or None for bytes

Warning: Do not use >1 byte encodings like UTF16 or UTF32 with CStrings.

Example:

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

`construct.`**`GreedyString`**(*encoding=None*)

A string that reads the rest of the stream until EOF, and writes a given string as is. If no encoding is given, this is essentially GreedyBytes.

Parameters **`encoding`** – encoding (e.g. "utf8") or None for bytes

See also:

Analog to `GreedyBytes` and the same when no enoding is used.

Example:

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

Core API: Structs and Sequences

`construct.Struct` (**subcons*, ***kw*)

A sequence of usually named constructs, similar to structs in C. The elements are parsed and built in the order they are defined.

Some fields do not need to be named, since they are built from None anyway. See Const Padding Pass Terminated.

See also:

Can be nested easily, and embedded using `Embedded()` wrapper that merges members into parent's members.

Parameters **subcons** – a sequence of subconstructs that make up this structure

Example:

```
>>> Struct("a"/Int8ul, "data"/Bytes(2), "data2"/Bytes(this.a)).parse(b"\x01abc")
Container(a=1)(data=b'ab')(data2=b'c')
>>> Struct("a"/Int8ul, "data"/Bytes(2), "data2"/Bytes(this.a)).build(_)
b'\x01abc'
>>> Struct("a"/Int8ul, "data"/Bytes(2), "data2"/Bytes(this.a)).build(dict(a=5,␣
↪data=b"??", data2=b"hello"))
b'\x05??hello'

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

Note that this syntax works ONLY on python 3.6 due to ordered keyword arguments:
>>> Struct(a=Byte, b=Byte, c=Byte, d=Byte)
```

`construct.Sequence` (**subcons*, ***kw*)

A sequence of unnamed constructs. The elements are parsed and built in the order they are defined.

See also:

Can be nested easily, and embedded using `Embedded()` wrapper that merges entries into parent's entries.

Parameters **subcons** – a sequence of subconstructs that make up this sequence

Example:

```
>>> (Byte >> Byte).build([1, 2])
b'\x01\x02'
>>> (Byte >> Byte).parse(_)
[1, 2]
>>> (Byte >> Byte).sizeof()
2

>>> Sequence(Byte, CString(), Float32b).build([255, b"hello", 123])
b'\xffhello\x00B\xf6\x00\x00'
>>> Sequence(Byte, CString(), Float32b).parse(_)
[255, b'hello', 123.0]
```

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

Embeds a struct into the enclosing struct, merging fields. Can also embed sequences into sequences. Name is also inherited.

Parameters **subcon** – the struct to embed

Example:

```
>>> Struct("a"/Byte, Embedded(Struct("b"/Byte)), "c"/Byte).parse(b"abc")
Container(a=97)(b=98)(c=99)
>>> Struct("a"/Byte, Embedded(Struct("b"/Byte)), "c"/Byte).build(_)
b'abc'
```

construct.**StopIf**(*condfunc*)

Checks for a condition, and stops a Struct Sequence Range from parsing or building.

Warning: May break sizeof methods. Unsure.

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)))
```

construct.**Union**(*parsefrom*, **subcons*, ***kw*)

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 if None, 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 the *parsefrom* parameter then stream will be left back at the starting offset. Many users fail to use this class properly.

Parameters

- **parsefrom** – how to leave stream after parsing, can be integer index or string name selecting a subcon, None (leaves stream at initial offset, the default), a context lambda returning either of previously mentioned

- **subcons** – subconstructs (order and name sensitive)

Example:

```
>>> Union(0, "raw"/Bytes(8), "ints"/Int32ub[2], "shorts"/Int16ub[4], "chars"/
↪Byte[8]).parse(b"12345678")
Container(raw=b'12345678')(ints=[825373492, 892745528])(shorts=[12594, 13108,␣
↪13622, 14136])(chars=[49, 50, 51, 52, 53, 54, 55, 56])

>>> Union(0, "raw"/Bytes(8), "ints"/Int32ub[2], "shorts"/Int16ub[4], "chars"/
↪Byte[8]).build(dict(chars=range(8)))
b'\x00\x01\x02\x03\x04\x05\x06\x07'

Note that this syntax works ONLY on python 3.6 due to ordered keyword arguments:
>>> Union(0, raw=Bytes(8), ints=Int32ub[2], shorts=Int16ub[4], chars=Byte[8])
>>> Union(0, raw=Bytes(8), ints=Int32ub[2], shorts=Int16ub[4], chars=Byte[8])
```

construct.**Select**(**subcons*, ***kw*)

Selects the first matching subconstruct. It will literally try each of the subconstructs, until one matches.

Parameters

- **subcons** – the subcons to try (order sensitive)
- **includename** – indicates whether to include the name of the selected subcon in the return value of parsing, default is false

Example:

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

Note that this syntax works ONLY on python 3.6 due to ordered keyword arguments:
>>> Select(num=Int32ub, text=CString(encoding="utf8"))
```

construct.**Optional**(*subcon*)

Makes an optional construct, that tries to parse the subcon. If parsing fails, returns None. If building fails, writes nothing.

Note: sizeof returns subcon size, although no bytes could be consumed or produced. Just something to consider.

Parameters **subcon** – the subcon to optionally parse or build

Example:

```
>>> Optional(Int64ul).parse(b"1234")
>>> Optional(Int64ul).parse(b"12345678")
4050765991979987505

>>> Optional(Int64ul).build(1)
b'\x01\x00\x00\x00\x00\x00\x00\x00'
>>> Optional(Int64ul).build("1")
b''
```

construct.**If**(*predicate*, *subcon*)

An if-then conditional construct. If the predicate indicates True, the *subcon* will be used for parsing and building, otherwise parsing returns None and building is no-op.

Parameters

- **predicate** – a function taking context and returning a bool
- **subcon** – the subcon that will be used if the predicate returns True

Example:

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

construct.**IfThenElse**(*predicate*, *thensubcon*, *elsesubcon*)

An if-then-else conditional construct. If the predicate indicates True, *thensubcon* will be used, otherwise *elsesubcon* will be used.

Parameters

- **predicate** – a function taking context and returning a bool
- **thensubcon** – the subcon that will be used if the predicate indicates True
- **elsesubcon** – the subcon that will be used if the predicate indicates False

Example:

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

construct.**Switch**(*keyfunc*, *cases*, *default=<NoDefault: None>*, *includekey=False*)

A conditional branch. Switch will choose the case to follow based on the return value of keyfunc. If no case is matched, and no default value is given, SwitchError will be raised.

Warning: You can use Embedded(Switch(...)) but not Switch(Embedded(...)). Sames applies to If and IfThenElse macros.

Parameters

- **keyfunc** – a context function that returns a key which will choose a case, or a constant
- **cases** – a dictionary mapping keys to subcons
- **default** – a default field to use when the key is not found in the cases. if not supplied, an exception will be raised when the key is not found. Pass can be used for do-nothing
- **includekey** – whether to include the key in the return value of parsing, defualt is False

Example:

```
>>> Switch(this.n, { 1:Byte, 2:Int32ub }).build(5, dict(n=1))
b'\x05'
>>> Switch(this.n, { 1:Byte, 2:Int32ub }).build(5, dict(n=2))
b'\x00\x00\x00\x05'
```

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

Makes a structure where each field is aligned to the same modulus.

See also:

Uses `Aligned()` and *~construct.core.Struct*.

Parameters

- **modulus** – passed to each member
- ***subcons** – the subcons that make up this structure
- **pattern** – optional, keyword parameter passed to each member

Example:

```
>>> AlignedStruct(4, "a"/Int8ub, "b"/Int16ub).build(dict(a=1,b=5))
b'\x01\x00\x00\x00\x00\x05\x00\x00'
>>> AlignedStruct(4, "a"/Int8ub, "b"/Int16ub).parse(_)
Container(a=1)(b=5)
>>> AlignedStruct(4, "a"/Int8ub, "b"/Int16ub).sizeof()
8
```

construct.**BitStruct**(**subcons*)

Makes a structure inside a Bitwise.

See also:

Uses `Bitwise()` and `Struct()`.

Parameters ***subcons** – the subcons that make up this structure

Example:

```
>>> BitStruct("field"/Octet).build(dict(field=5))
b'\x05'
>>> BitStruct("field"/Octet).parse(_)
Container(field=5)
>>> BitStruct("field"/Octet).sizeof()
1

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

construct.**EmbeddedBitStruct**(**subcons*)

Makes an embedded BitStruct.

See also:

Uses `Bitwise()` and `Embedded()` and `Struct()`.

Parameters ***subcons** – the subcons that make up this structure

Core API: Repeaters

`construct.`**`Array`**`(`*count*, *subcon*`)`

A homogenous array of elements. The array will iterate through exactly `count` elements. Will raise RangeError if less elements are found.

See also:

Base *`Range()`* construct.

Parameters

- **`count`** – int or a function that takes context and returns the number of elements
- **`subcon`** – the subcon to process individual elements

Example:

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

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

`construct.`**`PrefixedArray`**`(`*lengthfield*, *subcon*`)`

An array prefixed by a length field (as opposed to prefixed by byte count, see *`Prefixed()`*).

Parameters

- **`lengthfield`** – field parsing and building an integer
- **`subcon`** – subcon to process individual elements

Example:

```
>>> PrefixedArray(Byte, Byte).build(range(5))
b'\x05\x00\x01\x02\x03\x04'
>>> PrefixedArray(Byte, Byte).parse(_)
[0, 1, 2, 3, 4]
```

`construct.`**`Range`**`(`*min*, *max*, *subcon*`)`

A homogenous array of elements. The array will iterate through between `min` to `max` times. If an exception occurs (EOF, validation error), the repeater exits cleanly. If less than `min` units have been successfully parsed, a RangeError is raised.

See also:

Analog *`GreedyRange()`* that parses until end of stream.

Note: This object requires a seekable stream for parsing.

Parameters

- **`min`** – the minimal count

- **max** – the maximal count
- **subcon** – the subcon to process individual elements

Example:

```
>>> Range(3, 5, Byte).build([1,2,3,4])
b'\x01\x02\x03\x04'
>>> Range(3, 5, Byte).parse(_)
[1, 2, 3, 4]

>>> Range(3, 5, Byte).build([1,2])
construct.core.RangeError: expected from 3 to 5 elements, found 2
>>> Range(3, 5, Byte).build([1,2,3,4,5,6])
construct.core.RangeError: expected from 3 to 5 elements, found 6
```

construct.**GreedyRange**(*subcon*)

A homogenous array of elements that parses until end of stream and builds from all elements.

Parameters **subcon** – the subcon to process individual elements

Example:

```
>>> GreedyRange(Byte).build(range(10))
b'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t'
>>> GreedyRange(Byte).parse(_)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

construct.**RepeatUntil**(*predicate*, *subcon*)

An array that repeats until the predicate indicates it to stop. Note that the last element (which caused the repeat to exit) is included in the return value.

Parameters

- **predicate** – a predicate function that takes (obj, list, context) and returns True to break or False to continue
- **subcon** – the subcon used to parse and build each element

Example:

```
>>> RepeatUntil(lambda x,lst,ctx: x>7, Byte).build(range(20))
b'\x00\x01\x02\x03\x04\x05\x06\x07\x08'
>>> RepeatUntil(lambda x,lst,ctx: x>7, Byte).parse(b"\x01\xff\x02")
[1, 255]
>>> RepeatUntil(lambda x,lst,ctx: lst[-2:]==[0,0], Byte).parse(b"\x01\x00\x00\xff
↪")
[1, 0, 0]
```

Core API: Lazy equivalents

construct.**LazyStruct**(**subcons*, ***kw*)

Equivalent to Struct construct, however fixed size members are parsed on demand, others are parsed immediately. If entire struct is fixed size then entire parse is essentially one seek.

See also:

Equivalent to `Struct()`.

`construct.`**`LazySequence`**`(`**subcons*, ***kw*`)`

Equivalent to Sequence construct, however fixed size members are parsed on demand, others are parsed immediately. If entire sequence is fixed size then entire parse is essentially one seek.

See also:

Equivalent to `Sequence()`.

`construct.`**`LazyRange`**`(`*min*, *max*, *subcon*`)`

Equivalent to Range construct, but members are parsed on demand. Works only with fixed size subcon.

See also:

Equivalent to `Range()`.

`construct.`**`OnDemand`**`(`*subcon*`)`

Allows for on-demand (lazy) parsing. When parsing, it will return a parameterless function that when called, will return the parsed value. Object is cached after first parsing, so non-deterministic subcons will be affected. Works only with fixed size subcon.

Parameters **`subcon`** – the subcon to read/write on demand, must be fixed size

Example:

```
>>> OnDemand(Byte).parse(b"\xff")
<function OnDemand._parse.<locals>.<lambda> at 0x7fdc241cfc80>
>>> _()
255
>>> OnDemand(Byte).build(16)
b'\x10'

Can also re-build from the lambda returned at parsing.

>>> OnDemand(Byte).parse(b"\xff")
<function OnDemand._parse.<locals>.<lambda> at 0x7fcbd9855f28>
>>> OnDemand(Byte).build(_)
b'\xff'
```

`construct.`**`OnDemandPointer`**`(`*offset*, *subcon*`)`

An on-demand pointer. Is both lazy and jumps to a position before reading.

See also:

Base `OnDemand()` and `Pointer()` construct.

Parameters

- **`offset`** – an int or a context function that returns absolute stream position, where the construction would take place, can return negative integer as position from the end backwards
- **`subcon`** – the subcon that will be parsed or built at the *offset* stream position

Example:

```
>>> OnDemandPointer(lambda ctx: 2, Byte).parse(b"\x01\x02\x03garbage")
<function OnDemand._parse.<locals>.effectuate at 0x7f6f011ad510>
>>> _()
3
```

`construct.`**`LazyBound`**`(`*subconfunc*`)`

A lazy-bound construct that binds to the construct only at runtime. Useful for recursive data structures (like linked lists or trees), where a construct needs to refer to itself (while it doesn't exist yet).

Parameters **subconfunc** – a context function returning a Construct (derived) instance, can also return Pass or itself

Example:

```
>>> st = Struct(
...     "value"/Byte,
...     "next"/If(this.value > 0, LazyBound(lambda ctx: st)),
... )
...
>>> st.parse(b"\x05\x09\x00")
Container(value=5)(next=Container(value=9)(next=Container(value=0)(next=None)))
...
>>> print(st.parse(b"\x05\x09\x00"))
Container:
    value = 5
    next = Container:
        value = 9
        next = Container:
            value = 0
            next = None
```

Core API: Miscellaneous

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

Constant field enforcing a constant value. It is used for file signatures, to validate that the given pattern exists. When parsed, the value must match.

Note that a variable length 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. Consider for example, a field that eats null bytes and returns following byte, then compares to one. When parsing, both b"x00x00x01" and b"x01" will be parsed and checked OK.

Parameters

- **subcon** – the subcon used to build value from, or a b-string value itself
- **value** – optional, the expected value

Raises **ConstError** – when parsed data does not match specified value, or building from wrong value

Example:

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

>>> Const(Int32ul, 16).build(None)
b'\x10\x00\x00\x00'
```

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

A computed value. Underlying byte stream is unaffected. When parsing *func(context)* provides the value.

Parameters **func** – a function that takes context and returns the computed value

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'
```

```
>>> Computed(lambda ctx: os.urandom(10)).parse(b"")
b'\x98\xc2\xec\x10\x07\xf5\x8e\x98\xc2\xec'
```

construct.**Numpy**()

Preserves numpy arrays (both shape, dtype and values).

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 strings. Just like the standard namedtuple does.

Example:

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

construct.**Rebuild**(*subcon*, *func*)

Parses the field like normal, but computes the value for building from a function. Useful for length and count fields when Prefixed and PrefixedArray cannot be used.

Example:

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

construct.**Check**(*func*)

Checks for a condition, and raises ValidationError if the check fails.

Example:

```
Check(lambda ctx: len(ctx.payload.data) == ctx.payload_len)

Check(len_(this.payload.data) == this.payload_len)
```

`construct.`**`FocusedSeq`**(*parsebuildfrom*, **subcons*, ***kw*)

Parses and builds a sequence where only one subcon value is returned from parsing or taken into building, other fields are parsed and discarded or built from nothing. This is a replacement for SeqOfOne.

Parameters

- **`parsebuildfrom`** – which subcon to use, an int or str, or a context lambda returning an int or str
- **`*subcons`** – a list of members
- **`**kw`** – a list of members (works ONLY on python 3.6)

Excample:

```
>>> d = FocusedSeq("num", Const(b"MZ"), "num"/Byte, Terminated)
>>> d = FocusedSeq(1,     Const(b"MZ"), "num"/Byte, Terminated)

>>> d.parse(b"MZ\xff")
255
>>> d.build(255)
b'MZ\xff'
```

`construct.`**`Default`**(*subcon*, *value*)

Allows to make a field have a default value, which comes handly when building a Struct from a dict with missing keys.

Example:

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

Core API: Alignment and Padding

`construct.`**`Padding`**(*length*, *pattern='\x00'*, *strict=False*)

A padding field that adds bytes when building, discards bytes when parsing.

Parameters

- **`length`** – length of the padding, an int or a function taking context and returning an int
- **`pattern`** – padding pattern as b-string character, default is b"x00" null character
- **`strict`** – whether to verify during parsing that the stream contains the pattern, raises an exception if actual padding differs from the pattern, default is False

Example:

```
>>> (Padding(4) >> Bytes(4)).parse(b"????abcd")
[None, b'abcd']
>>> (Padding(4) >> Bytes(4)).build(_)
```

```
b'\x00\x00\x00\x00abcd'
>>> (Padding(4) >> Bytes(4)).sizeof()
8

>>> Padding(4).build(None)
b'\x00\x00\x00\x00'
>>> Padding(4, strict=True).parse(b"****")
construct.core.PaddingError: expected b'\x00\x00\x00\x00', found b'****'
```

`construct.`**`Padded`**(*length*, *subcon*, *pattern='\x00'*, *strict=False*)

Appends additional null bytes to achieve a fixed length.

Example:

```
>>> Padded(4, Byte).build(255)
b'\xff\x00\x00\x00'
>>> Padded(4, Byte).parse(_)
255
>>> Padded(4, Byte).sizeof()
4

>>> Padded(4, VarInt).build(1)
b'\x01\x00\x00\x00'
>>> Padded(4, VarInt).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.

Parameters

- **`modulus`** – the modulus to final length, an int or a context->int function
- **`subcon`** – the subcon to align
- **`pattern`** – optional, the padding pattern (default is x00)

Example:

```
>>> Aligned(4, Int16ub).build(1)
b'\x00\x01\x00\x00'
>>> Aligned(4, Int16ub).parse(_)
1
>>> Aligned(4, Int16ub).sizeof()
4
```

Core API: Streaming

`construct.`**`Pointer`**(*offset*, *subcon*)

Changes the stream position to a given offset, where the construction should take place, and restores the stream position when finished.

See also:

Analog *`OnDemandPointer()`* field, which also seeks to a given offset.

Parameters

- **offset** – an int or a function that takes context and returns absolute stream position, where the construction would take place, can return negative integer as position from the end backwards
- **subcon** – the subcon to use at the offset

Example:

```
>>> Pointer(8, Bytes(1)).parse(b"abcdefghijkl")
b'i'
>>> Pointer(8, Bytes(1)).build(b"x")
b'\x00\x00\x00\x00\x00\x00\x00\x00x'
>>> Pointer(8, Bytes(1)).sizeof()
0
```

`construct.`**`Peek`**`(`*subcon*`)`

Peeks at the stream. Parses without changing the stream position. If the end of the stream is reached when peeking, returns None. Sizeof returns 0 by design because build does not put anything into the stream. Building is no-op.

See also:

The *`Union()`* class.

Parameters **subcon** – the subcon to peek at

Example:

```
>>> Sequence(Peek(Byte), Peek(Int16ub)).parse(b"\x01\x02")
[1, 258]
>>> Sequence(Peek(Byte), Peek(Int16ub)).sizeof()
0
```

`construct.`**`Tell`**`()`

Gets the stream position when parsing or building.

Tells are 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.

See also:

Better use *`RawCopy()`* instead of manually extracting bytes.

Example:

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

`construct.`**`Seek`**`(`*at*, *whence=0*`)`

Sets a new stream position when parsing or building. Seeks are useful when many other fields follow the jump. Pointer works when there is only one field to look at, but when there is more to be done, Seek may come useful.

See also:

Analog *`Pointer()`* wrapper that has same side effect but also processed a subcon.

Parameters

- **at** – where to jump to, can be an int or a context lambda
- **whence** – is the offset from beginning (0) or from current position (1) or from ending (2), can be an int or a context lambda, default is 0

Example:

```
>>> (Seek(5) >> Byte).parse(b"01234x")
[5, 120]
>>> (Bytes(10) >> Seek(5) >> Byte).build([b"0123456789", None, 255])
b'01234\xff6789'
```

construct.**Pass**()

A do-nothing construct, useful as the default case for Switch. Returns None on parsing, puts nothing on building.

Example:

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

construct.**Terminated**()

Asserts the end of the stream has been reached at the point it was placed. You can use this to ensure no more unparsed data follows.

This construct is only meaningful for parsing. For building, it's a no-op.

Example:

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

construct.**Restreamed**(*subcon*, *encoder*, *encoderunit*, *decoder*, *decoderunit*, *decoderunitname*, *sizecomputer*)

Transforms bytes between the underlying stream and the subcon.

When the parsing or building is done, the wrapper stream is closed. If read buffer or write buffer is not empty, error is raised.

See also:

Both *Bitwise()* and *Bytewise()* are implemented using Restreamed.

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 after parsing/building. Also do NOT use pointers inside.

Parameters

- **subcon** – the subcon which will operate on the buffer
- **encoder** – a function that takes a b-string and returns a b-string (used when building)
- **encoderunit** – ratio as int, encoder takes that many bytes at once

- **decoder** – a function that takes a b-string and returns a b-string (used when parsing)
- **decoderunit** – ratio as int, decoder takes that many bytes at once
- **decoderunitname** – English string that describes the units (plural) returned by the decoder. Used for error messages.
- **sizecomputer** – a function that computes amount of bytes outputed by some bytes

Example:

```
Bitwise  <--> Restreamed(subcon, bits2bytes, 8, bytes2bits, 1, lambda n: n//8)
Bytewise <--> Restreamed(subcon, bytes2bits, 1, bits2bytes, 8, lambda n: n*8)
```

construct.**Rebuffered**(*subcon*, *tailcutoff=None*)

Caches bytes from the underlying stream, so it becomes seekable and tellable. Also makes the stream blocking, in case it came from a socket or a pipe. Optionally, stream can forget bytes that went a certain amount of bytes beyond the current offset, allowing only a limited seeking capability while allowing to process an endless stream.

Warning: Experimental implementation. May not be mature enough.

Parameters

- **subcon** – the subcon which will operate on the buffered stream
- **tailcutoff** – optional, amount of bytes kept in buffer, by default buffers everything

Example:

```
Rebuffered(RepeatUntil(lambda obj,ctx: ?,Byte), tailcutoff=1024).parse_
↪stream(endless_nonblocking_stream)
```

Core API: Tunneling

construct.**RawCopy**(*subcon*)

Returns a dict containing both parsed subcon, the raw bytes that were consumed by it, starting and ending offset in the stream, and the amount of bytes. Builds either from raw bytes or a value used by subcon.

Context does contain a dict with data (if built from raw bytes) or with both (if built from value or parsed).

Example:

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

construct.**ByteSwapped**(*subcon*)

Swap the byte order within boundaries of the given subcon.

Parameters **subcon** – the subcon on top of byte swapped bytes

Example:

```
Int24ul <--> ByteSwapped(Int24ub)
```

`construct.`**`BitsSwapped`**`(`*subcon*`)`

Swap the bit order within each byte within boundaries of the given subcon.

Parameters **`subcon`** – the subcon on top of byte swapped bytes

Example:

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

`construct.`**`Prefixed`**`(`*lengthfield*, *subcon*, *includelength=False*`)`

Parses the length field. Then reads that amount of bytes and parses the subcon using only those bytes. Constructs that consume entire remaining stream are constrained to consuming only the specified amount of bytes. When building, data is prefixed by its length. Optionally, length field can include its own size.

See also:

The `VarInt` encoding should be preferred over `Byte` and fixed size fields. VarInt is more compact and does never overflow.

Parameters

- **`lengthfield`** – a subcon used for storing the length
- **`subcon`** – the subcon used for storing the value
- **`includelength`** – optional, whether length field should include own size

Example:

```
>>> Prefixed(VarInt, GreedyBytes).parse(b"\x05hello?????")
b'hello'

>>>> Prefixed(VarInt, Byte[:]).parse(b"\x03\x01\x02\x03?????")
[1, 2, 3]
```

`construct.`**`Checksum`**`(`*checksumfield*, *hashfunc*, *bytesfunc*`)`

A field that is build or validated by a hash of a given byte range.

Parameters

- **`checksumfield`** – a subcon field that reads the checksum, usually Bytes(int)
- **`hashfunc`** – a function taking bytes and returning whatever checksumfield takes when building
- **`bytesfunc`** – a function taking context and returning the bytes or object to be hashed, usually this.rawcopy1.data alike

Example:

```
import hashlib
d = Struct(
    "fields" / RawCopy(Struct(
        "a" / Byte,
        "b" / Byte,
    )),
    "checksum" / Checksum(Bytes(64), lambda data: hashlib.sha512(data).digest(),␣
↪this.fields.data),
)
data = d.build(dict(fields=dict(value=dict(a=1,b=2))))
# returned b'\x01\x02\xbd\xd8\x1a\xb23\xbc\xebj\xd23\xcd\x18qP\x93␣
↪\xa1\x8d\x035\xa8\x91\xcf\x98s\t\x90\xe8\x92>\x1d\xda\x04\xf35\x8e\x9c~
↪\x1c=\x16\xb1o@\x8c\xfa\xfbj\xf52T\xef0#\xed$6S8\x08\xb6\xca\x993'
```

construct.**Compressed**(*subcon*, *encoding*, *level=None*)

Compresses and decompresses underlying stream when processing the subcon. 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.

Parameters

- **subcon** – the subcon used for storing the value
- **encoding** – any of the module names like zlib/gzip/bzip2/lzma, otherwise any of codecs module bytes<->bytes encodings
- **level** – optional, an int between 0..9, lzma discards it

Example:

```
Compressed(GreedyBytes, "zlib")

Prefixed(VarInt, Compressed(GreedyBytes, "zlib"))
Struct("inner"/above)

Compressed(Struct(...), "zlib")
```

Core API: Debugging

Debugging utilities for constructs

class construct.debug.**Debugger**(*subcon*)

A 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** – the subcon to debug

Example:

```
>>> Debugger(Byte[3]).build([])
===============================================================================
Debugging exception of <Range: None>:
  File "/home/arkadiusz/Dokumenty/GitHub/construct/construct/debug.py", line 116,␣
↪in _build
    obj.stack.append(a)
  File "/home/arkadiusz/Dokumenty/GitHub/construct/construct/core.py", line 1069,␣
↪in _build
    raise RangeError("expected from %d to %d elements, found %d" % (self.min,␣
↪self.max, len(obj)))
construct.core.RangeError: expected from 3 to 3 elements, found 0

> /home/arkadiusz/Dokumenty/GitHub/construct/construct/core.py(1069)_build()
-> raise RangeError("expected from %d to %d elements, found %d" % (self.min, self.
↪max, len(obj)))
```

```
(Pdb)
================================================================================

>>> format = Struct(
...     "spam" / Debugger(Enum(Byte, A=1, B=2, C=3)),
... )
>>> format.parse(b"\xff")
================================================================================
Debugging exception of <Mapping: None>:
  File "/home/arkadiusz/Dokumenty/GitHub/construct/construct/core.py", line 2578,␣
↪in _decode
    return self.decoding[obj]
KeyError: 255

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "/home/arkadiusz/Dokumenty/GitHub/construct/construct/debug.py", line 127,␣
↪in _parse
    return self.subcon._parse(stream, context)
  File "/home/arkadiusz/Dokumenty/GitHub/construct/construct/core.py", line 308,␣
↪in _parse
    return self._decode(self.subcon._parse(stream, context), context)
  File "/home/arkadiusz/Dokumenty/GitHub/construct/construct/core.py", line 2583,␣
↪in _decode
    raise MappingError("no decoding mapping for %r" % (obj,))
construct.core.MappingError: no decoding mapping for 255

(you can set the value of 'self.retval', which will be returned)
> /home/arkadiusz/Dokumenty/GitHub/construct/construct/core.py(2583)_decode()
-> raise MappingError("no decoding mapping for %r" % (obj,))
(Pdb) self.retval = "???"
(Pdb) q
```

class `construct.debug.`**`Probe`**(*name=None*, *show_stream=True*, *show_context=True*, *show_stack=True*, *stream_lookahead=128*, *func=None*)

A probe: dumps the context, stack frames, and stream content to the screen to aid the debugging process.

Parameters

- **`name`** – the display name
- **`show_stream`** – whether or not to show stream contents. default is True. the stream must be seekable.
- **`show_context`** – whether or not to show the context. default is True.
- **`show_stack`** – whether or not to show the upper stack frames. default is True.
- **`stream_lookahead`** – the number of bytes to dump when show_stack is set. default is 100.

Example:

```
>>> Struct("count"/Byte, "items"/Byte[this.count], Probe()).parse(b"\x05abcde")
================================================================================
Probe <unnamed 3>
EOF reached
Container:
    count = 5
    items = ListContainer:
        97
        98
        99
        100
        101
================================================================================
Container(count=5)(items=[97, 98, 99, 100, 101])

>>> (Byte >> Probe()).parse(b"?")
================================================================================
Probe <unnamed 1>
EOF reached
Container:
    0 = 63
================================================================================
[63, None]
```

`construct.debug.ProbeInto(func)`

ProbeInto looks inside the context and extracts a part of it using a lambda instead of printing the entire context.

Example:

```
>>> st = "junk"/RepeatUntil(obj_ == 0,Byte) + "num"/Byte + Probe()
>>> st.parse(b"xcnzxmbjskahuiwerhquiehnsdjk\x00\xff")
================================================================================
Probe <unnamed 5>
path is parsing
EOF reached
Container:
    junk = ListContainer:
        120
        99
        110
        122
        120
        109
        98
        106
        115
        107
        97
        104
        117
        105
        119
        101
        114
        104
        113
        117
        105
        101
        104
        110
        115
        100
        106
        107
        0
    num = 255
================================================================================
Container(junk=[120, 99, 110, 122, 120, 109, 98, 106, 115, 107, 97, 104, 117, 105,
↪ 119, 101, 114, 104, 113, 117, 105, 101, 104, 110, 115, 100, 106, 107,␣
↪0])(num=255)

>>> st = "junk"/RepeatUntil(obj_ == 0,Byte) + "num"/Byte + ProbeInto(this.num)
>>> st.parse(b"xcnzxmbjskahuiwerhquiehnsdjk\x00\xff")
================================================================================
Probe <unnamed 6>
path is parsing
EOF reached
255
================================================================================
Container(junk=[120, 99, 110, 122, 120, 109, 98, 106, 115, 107, 97, 104, 117, 105,
↪ 119, 101, 114, 104, 113, 117, 105, 101, 104, 110, 115, 100, 106, 107,␣
↪0])(num=255)
```

construct.**Error**()

Raises an exception when triggered by parse or build. Can be used as a sentinel that blows a whistle when a conditional branch goes the wrong way, or to raise an error explicitly the declarative way.

Example:

```
>>> d = "x"/Int8sb >> IfThenElse(this.x > 0, Int8sb, Error)
>>> d.parse(b"\xff\x05")
construct.core.ExplicitError: Error field was activated during parsing
```

construct.core – entire module

class construct.core.**Adapter**(*subcon*)

Abstract adapter parent class.

Needs to implement `_decode()` and `_encode()`.

Parameters **subcon** – the construct to wrap

class construct.core.**Aligned**(*modulus*, *subcon*, *pattern='x00'*)

Appends additional null bytes to achieve a length that is shortest multiple of a modulus.

Parameters

- **modulus** – the modulus to final length, an int or a context->int function
- **subcon** – the subcon to align
- **pattern** – optional, the padding pattern (default is x00)

Example:

```
>>> Aligned(4, Int16ub).build(1)
b'\x00\x01\x00\x00'
>>> Aligned(4, Int16ub).parse(_)
1
>>> Aligned(4, Int16ub).sizeof()
4
```

construct.core.**AlignedStruct**(*modulus*, **subcons*, ***kw*)

Makes a structure where each field is aligned to the same modulus.

See also:

Uses `Aligned()` and *~construct.core.Struct*.

Parameters

- **modulus** – passed to each member
- ***subcons** – the subcons that make up this structure
- **pattern** – optional, keyword parameter passed to each member

Example:

```
>>> AlignedStruct(4, "a"/Int8ub, "b"/Int16ub).build(dict(a=1,b=5))
b'\x01\x00\x00\x00\x00\x05\x00\x00'
>>> AlignedStruct(4, "a"/Int8ub, "b"/Int16ub).parse(_)
Container(a=1)(b=5)
>>> AlignedStruct(4, "a"/Int8ub, "b"/Int16ub).sizeof()
8
```

construct.core.**Array**(*count*, *subcon*)

A homogenous array of elements. The array will iterate through exactly `count` elements. Will raise RangeError if less elements are found.

See also:

Base `Range()` construct.

Parameters

- **count** – int or a function that takes context and returns the number of elements
- **subcon** – the subcon to process individual elements

Example:

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

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

construct.core.**BitStruct**(**subcons*)

Makes a structure inside a Bitwise.

See also:

Uses `Bitwise()` and `Struct()`.

Parameters ***subcons** – the subcons that make up this structure

Example:

```
>>> BitStruct("field"/Octet).build(dict(field=5))
b'\x05'
>>> BitStruct("field"/Octet).parse(_)
Container(field=5)
>>> BitStruct("field"/Octet).sizeof()
1

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

class `construct.core.`**`BitsInteger`**(*length*, *signed=False*, *swapped=False*, *bytesize=8*)

A byte field, that parses into and builds from integers as opposed to b-strings. This is similar to Bit/Nibble/Octet fields but can have arbitrary sizes. This must be enclosed in Bitwise.

Parameters

- **`length`** – number of bits in the field, or a context function that returns int
- **`signed`** – whether the value is signed (two's complement), default is False (unsigned)
- **`swapped`** – whether to swap byte order (little endian), default is False (big endian)
- **`bytesize`** – size of byte as used for byte swapping (if swapped), default is 8

Example:

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

`construct.core.`**`BitsSwapped`**(*subcon*)

Swap the bit order within each byte within boundaries of the given subcon.

Parameters **`subcon`** – the subcon on top of byte swapped bytes

Example:

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

`construct.core.`**`Bitwise`**(*subcon*)

Converts the stream from bytes to bits, and passes the bitstream to underlying subcon.

See also:

Analog *`Bytewise()`* that transforms subset of bits back to bytes.

Warning: Do not use pointers inside.

Parameters **subcon** – any field that works with bits like: BitStruct BitsNumber Bit Nibble Octet

Example:

```
>>> Bitwise(Octet).parse(b"\xff")
255
>>> Bitwise(Octet).build(1)
b'\x01'
>>> Bitwise(Octet).sizeof()
1
```

construct.core.**ByteSwapped**(*subcon*)

Swap the byte order within boundaries of the given subcon.

Parameters **subcon** – the subcon on top of byte swapped bytes

Example:

```
Int24ul <--> ByteSwapped(Int24ub)
```

class construct.core.**Bytes**(*length*)

A field consisting of a specified number of bytes. Builds from a b-string, or an integer (although deprecated and BytesInteger should be used).

See also:

Analog *BytesInteger()* that parses and builds from integers.

Parameters **length** – an int or a function that takes context and returns int

Example:

```
>>> Bytes(4).parse(b"beef")
b'beef'
>>> Bytes(4).build(_)
b'beef'
>>> Bytes(4).build(255)
b'\x00\x00\x00\xff'
>>> Bytes(4).sizeof()
4
```

class construct.core.**BytesInteger**(*length*, *signed=False*, *swapped=False*, *bytesize=1*)

A byte field, that parses into and builds from integers as opposed to b-strings. This is similar to Int* fields but can have arbitrary size.

See also:

Analog *BitsInteger()* that operates on bits.

Parameters

- **length** – number of bytes in the field, or a function that takes context and returns int
- **signed** – whether the value is signed (two's complement), default is False (unsigned)
- **swapped** – whether to swap byte order (little endian), default is False (big endian)

- **bytesize** – size of byte as used for byte swapping (if swapped), default is 1

Example:

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

construct.core.**Bytewise**(*subcon*)
Converts the stream from bits back to bytes. Needs to be used within Bitwise.

Parameters **subcon** – any field that works with bytes like: Bytes BytesInteger Int* Struct

Example:

```
>>> Bitwise(Bytewise(Byte)).parse(b"\xff")
255
>>> Bitwise(Bytewise(Byte)).build(63)
b'?'
>>> Bitwise(Bytewise(Byte)).sizeof()
1
```

construct.core.**CString**(*terminators='\x00', encoding=None*)
A string ending in a terminator b-string character.

CString is similar to the strings of C.

By default, the terminator is the NULL byte (b'x00'). Terminators field can be a longer b-string, and any of the characters breaks parsing. First terminator byte is used when building.

Parameters

- **terminators** – sequence of valid terminators, first is used when building, all are used when parsing
- **encoding** – encoding (e.g. "utf8") or None for bytes

Warning: Do not use >1 byte encodings like UTF16 or UTF32 with CStrings.

Example:

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

class construct.core.**Check**(*func*)
Checks for a condition, and raises ValidationError if the check fails.

Example:

```
Check(lambda ctx: len(ctx.payload.data) == ctx.payload_len)

Check(len_(this.payload.data) == this.payload_len)
```

class construct.core.**Checksum**(*checksumfield*, *hashfunc*, *bytesfunc*)

A field that is build or validated by a hash of a given byte range.

Parameters

- **checksumfield** – a subcon field that reads the checksum, usually Bytes(int)
- **hashfunc** – a function taking bytes and returning whatever checksumfield takes when building
- **bytesfunc** – a function taking context and returning the bytes or object to be hashed, usually this.rawcopy1.data alike

Example:

```
import hashlib
d = Struct(
    "fields" / RawCopy(Struct(
        "a" / Byte,
        "b" / Byte,
    )),
    "checksum" / Checksum(Bytes(64), lambda data: hashlib.sha512(data).digest(),␣
↪this.fields.data),
)
data = d.build(dict(fields=dict(value=dict(a=1,b=2))))
# returned b'\x01\x02\xbd\xd8\x1a\xb23\xbc\xebj\xd23\xcd\x18qP\x93␣
↪\xa1\x8d\x035\xa8\x91\xcf\x98s\t\x90\xe8\x92>\x1d\xda\x04\xf35\x8e\x9c~
↪\x1c=\x16\xb1o@\x8c\xfa\xfbj\xf52T\xef0#\xed$6S8\x08\xb6\xca\x993'
```

class construct.core.**Compressed**(*subcon*, *encoding*, *level=None*)

Compresses and decompresses underlying stream when processing the subcon. 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.

Parameters

- **subcon** – the subcon used for storing the value
- **encoding** – any of the module names like zlib/gzip/bzip2/lzma, otherwise any of codecs module bytes<->bytes encodings
- **level** – optional, an int between 0..9, lzma discards it

Example:

```
Compressed(GreedyBytes, "zlib")

Prefixed(VarInt, Compressed(GreedyBytes, "zlib"))
Struct("inner"/above)

Compressed(Struct(...), "zlib")
```

class construct.core.**Computed**(*func*)

A computed value. Underlying byte stream is unaffected. When parsing *func(context)* provides the value.

Parameters **func** – a function that takes context and returns the computed value

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'
```

```
>>> Computed(lambda ctx: os.urandom(10)).parse(b"")
b'\x98\xc2\xec\x10\x07\xf5\x8e\x98\xc2\xec'
```

class `construct.core.`**`Const`**(*subcon*, *value=None*)

Constant field enforcing a constant value. It is used for file signatures, to validate that the given pattern exists. When parsed, the value must match.

Note that a variable length 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. Consider for example, a field that eats null bytes and returns following byte, then compares to one. When parsing, both b"x00x00x01" and b"x01" will be parsed and checked OK.

Parameters

- **`subcon`** – the subcon used to build value from, or a b-string value itself
- **`value`** – optional, the expected value

Raises `ConstError` – when parsed data does not match specified value, or building from wrong value

Example:

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

>>> Const(Int32ul, 16).build(None)
b'\x10\x00\x00\x00'
```

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.

The external user API:

- `parse()`
- `parse_stream()`
- `build()`
- `build_stream()`
- `sizeof()`

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

- `_parse()`
- `_build()`

- `_sizeof()`

And stateful copying:

- `__getstate__()`
- `__setstate__()`

All constructs have a name and flags. The name is used for naming struct members and context dictionaries. Note that the name can either be a string, or None if the name is not needed. A single underscore ("_") is a reserved name. 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 behavior.

`build`(*obj*, *context=None*, ***kw*)
Build an object in memory.

Returns bytes

`build_stream`(*obj*, *stream*, *context=None*, ***kw*)
Build an object directly into a stream.

Returns None

`parse`(*data*, *context=None*, ***kw*)
Parse an in-memory buffer.

Strings, buffers, memoryviews, and other complete buffers can be parsed with this method.

`parse_stream`(*stream*, *context=None*, ***kw*)
Parse a stream.

Files, pipes, sockets, and other streaming sources of data are handled by this method.

`sizeof`(*context=None*, ***kw*)
Calculate the size of this object, optionally using a context.

Some constructs have no fixed size and can only know their size for a given hunk of data. These constructs will raise an error if they are not passed a context.

Parameters **`context`** – a container

Returns int of the length of this construct

Raises **`SizeofError`** – the size could not be determined

class `construct.core.`**`Default`**(*subcon*, *value*)
Allows to make a field have a default value, which comes handly when building a Struct from a dict with missing keys.

Example:

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

class `construct.core.`**`Embedded`**(*subcon*)
Embeds a struct into the enclosing struct, merging fields. Can also embed sequences into sequences. Name is also inherited.

Parameters **`subcon`** – the struct to embed

Example:

```
>>> Struct("a"/Byte, Embedded(Struct("b"/Byte)), "c"/Byte).parse(b"abc")
Container(a=97)(b=98)(c=99)
>>> Struct("a"/Byte, Embedded(Struct("b"/Byte)), "c"/Byte).build(_)
b'abc'
```

`construct.core.`**`EmbeddedBitStruct`**`(`**subcons*`)`

Makes an embedded BitStruct.

See also:

Uses *`Bitwise()`* and *`Embedded()`* and *`Struct()`*.

Parameters **`*subcons`** – the subcons that make up this structure

`construct.core.`**`Enum`**`(`*subcon*, *default=NotImplemented*, ***mapping*`)`

A set of named values mapping. Can build both from names and values.

Parameters

- **`subcon`** – the subcon to map
- **`**mapping`** – keyword arguments which serve as the encoding mapping
- **`default`** – an optional, keyword-only argument that specifies the default value to use when the mapping is undefined. if not given, and exception is raised when the mapping is undefined. use *Pass* topass the unmapped value as-is

Example:

```
>>> Enum(Byte,a=1,b=2).parse(b"\x01")
'a'
>>> Enum(Byte,a=1,b=2).parse(b"\x08")
construct.core.MappingError: no decoding mapping for 8

>>> Enum(Byte,a=1,b=2).build("a")
b'\x01'
>>> Enum(Byte,a=1,b=2).build(1)
b'\x01'
```

class `construct.core.`**`ExprAdapter`**`(`*subcon*, *encoder*, *decoder*`)`

A generic adapter that takes `encoder` and `decoder` as parameters. You can use ExprAdapter instead of writing a full-blown class when only a simple expression is needed.

Parameters

- **`subcon`** – the subcon to adapt
- **`encoder`** – a function that takes (obj, context) and returns an encoded version of obj, or None for identity
- **`decoder`** – a function that takes (obj, context) and returns an decoded version of obj, or None for identity

Example:

```
Ident = ExprAdapter(Byte,
    encoder = lambda obj,ctx: obj+1,
    decoder = lambda obj,ctx: obj-1, )
```

class `construct.core.`**`ExprValidator`**(*subcon*, *validator*)

A generic adapter that takes `validator` as parameter. You can use ExprValidator instead of writing a full-blown class when only a simple expression is needed.

Parameters

- **`subcon`** – the subcon to adapt
- **`encoder`** – a function that takes (obj, context) and returns a bool

Example:

```
OneOf = ExprValidator(Byte,
    validator = lambda obj,ctx: obj in [1,3,5])
```

`construct.core.`**`Filter`**(*predicate*, *subcon*)

Filters a list leaving only the elements that passed through the validator.

Parameters

- **`subcon`** – a construct to validate, usually a Range or Array or Sequence
- **`predicate`** – a function taking (obj, context) and returning a bool

Example:

```
>>> Filter(obj_ != 0, Byte[:]).parse(b"\x00\x02\x00")
[2]
>>> Filter(obj_ != 0, Byte[:]).build([0,1,0,2,0])
b'\x01\x02'
```

class `construct.core.`**`FlagsEnum`**(*subcon*, ***flags*)

A set of flag values mapping. Each flag is extracted from the number, resulting in a FlagsContainer dict that has each key assigned True or False.

Parameters

- **`subcon`** – the subcon to extract
- **`**flags`** – a dictionary mapping flag-names to their value

Example:

```
>>> FlagsEnum(Byte,a=1,b=2,c=4,d=8).parse(b"\x03")
Container(c=False)(b=True)(a=True)(d=False)
>>> FlagsEnum(Byte,a=1,b=2,c=4,d=8).build(_)
b'\x03'
```

class `construct.core.`**`FocusedSeq`**(*parsebuildfrom*, **subcons*, ***kw*)

Parses and builds a sequence where only one subcon value is returned from parsing or taken into building, other fields are parsed and discarded or built from nothing. This is a replacement for SeqOfOne.

Parameters

- **`parsebuildfrom`** – which subcon to use, an int or str, or a context lambda returning an int or str
- **`*subcons`** – a list of members
- **`**kw`** – a list of members (works ONLY on python 3.6)

Excample:

```
>>> d = FocusedSeq("num", Const(b"MZ"), "num"/Byte, Terminated)
>>> d = FocusedSeq(1,     Const(b"MZ"), "num"/Byte, Terminated)

>>> d.parse(b"MZ\xff")
255
>>> d.build(255)
b'MZ\xff'
```

class `construct.core.`**`FormatField`**`(`*endianity*, *format*`)`

A field that uses `struct` module to pack and unpack data. This is used to implement basic Int* fields.

See `struct` documentation for instructions on crafting format strings.

Parameters

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

Example:

```
>>> FormatField(">","H").parse(b"\x01\x00")
256
>>> FormatField(">","H").build(18)
b'\x00\x12'
>>> FormatField(">","H").sizeof()
2
```

`construct.core.`**`GreedyRange`**`(`*subcon*`)`

A homogenous array of elements that parses until end of stream and builds from all elements.

Parameters **`subcon`** – the subcon to process individual elements

Example:

```
>>> GreedyRange(Byte).build(range(10))
b'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t'
>>> GreedyRange(Byte).parse(_)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

`construct.core.`**`GreedyString`**`(`*encoding=None*`)`

A string that reads the rest of the stream until EOF, and writes a given string as is. If no encoding is given, this is essentially GreedyBytes.

Parameters **`encoding`** – encoding (e.g. "utf8") or None for bytes

See also:

Analog to `GreedyBytes` and the same when no enoding is used.

Example:

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

`construct.core.`**`Hex`**`(`*subcon*`)`

Adapter for hex-dumping b-strings. It returns a hex dump when parsing, and un-dumps when building.

Example:

```
>>> Hex(GreedyBytes).parse(b"abcd")
b'61626364'
>>> Hex(GreedyBytes).build("01020304")
b'\x01\x02\x03\x04'
```

`construct.core.`**`HexDump`**(*subcon*, *linesize=16*)

Adapter for hex-dumping b-strings. It returns a hex dump when parsing, and un-dumps when building.

Parameters

- **`linesize`** – default 16 bytes per line
- **`buildraw`** – by default build takes the same format that parse returns, set to build from a b-string directly

Example:

```
>>> HexDump(Bytes(10)).parse(b"12345abc;/")
'0000   31 32 33 34 35 61 62 63 3b 2f                    12345abc;/       \n'
```

`construct.core.`**`If`**(*predicate*, *subcon*)

An if-then conditional construct. If the predicate indicates True, the *subcon* will be used for parsing and building, otherwise parsing returns None and building is no-op.

Parameters

- **`predicate`** – a function taking context and returning a bool
- **`subcon`** – the subcon that will be used if the predicate returns True

Example:

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

`construct.core.`**`IfThenElse`**(*predicate*, *thensubcon*, *elsesubcon*)

An if-then-else conditional construct. If the predicate indicates True, *thensubcon* will be used, otherwise *elsesubcon* will be used.

Parameters

- **`predicate`** – a function taking context and returning a bool
- **`thensubcon`** – the subcon that will be used if the predicate indicates True
- **`elsesubcon`** – the subcon that will be used if the predicate indicates False

Example:

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

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`** – the subcon to index

- **count** – expected number of elements, needed during building
- **index** – the index of the list to get
- **empty** – value to fill the list with during building

Example:

```
???
```

class construct.core.**LazyBound**(*subconfunc*)

A lazy-bound construct that binds to the construct only at runtime. Useful for recursive data structures (like linked lists or trees), where a construct needs to refer to itself (while it doesn't exist yet).

Parameters **subconfunc** – a context function returning a Construct (derived) instance, can also return Pass or itself

Example:

```
>>> st = Struct(
...     "value"/Byte,
...     "next"/If(this.value > 0, LazyBound(lambda ctx: st)),
... )
...
>>> st.parse(b"\x05\x09\x00")
Container(value=5)(next=Container(value=9)(next=Container(value=0)(next=None)))
...
>>> print(st.parse(b"\x05\x09\x00"))
Container:
    value = 5
    next = Container:
        value = 9
        next = Container:
            value = 0
            next = None
```

class construct.core.**LazyRange**(*min*, *max*, *subcon*)

Equivalent to Range construct, but members are parsed on demand. Works only with fixed size subcon.

See also:

Equivalent to *Range()*.

class construct.core.**LazySequence**(**subcons*, ***kw*)

Equivalent to Sequence construct, however fixed size members are parsed on demand, others are parsed immediately. If entire sequence is fixed size then entire parse is essentially one seek.

See also:

Equivalent to *Sequence()*.

class construct.core.**LazyStruct**(**subcons*, ***kw*)

Equivalent to Struct construct, however fixed size members are parsed on demand, others are parsed immediately. If entire struct is fixed size then entire parse is essentially one seek.

See also:

Equivalent to *Struct()*.

class construct.core.**Mapping**(*subcon*, *decoding*, *encoding*, *decdefault=NotImplemented*, *encdefault=NotImplemented*)

Adapter that maps objects to other objects. Translates objects before parsing and before

Parameters

- **subcon** – the subcon to map
- **decoding** – the decoding (parsing) mapping as a dict
- **encoding** – the encoding (building) mapping as a dict
- **decdefault** – the default return value when object is not found in the mapping, if no object is given an exception is raised, if `Pass` is used, the unmapped object will be passed as-is
- **encdefault** – the default return value when object is not found in the mapping, if no object is given an exception is raised, if `Pass` is used, the unmapped object will be passed as-is

Example:

```
???
```

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 strings. Just like the standard namedtuple does.

Example:

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

`construct.core.`**`NoneOf`**(*subcon*, *invalids*)

Validates that the object is none of the listed values, both during parsing and building.

Parameters

- **subcon** – a construct to validate
- **invalids** – a collection implementing *in*

See also:

Look at `OneOf()` for examples, works the same.

class `construct.core.`**`OnDemand`**(*subcon*)

Allows for on-demand (lazy) parsing. When parsing, it will return a parameterless function that when called, will return the parsed value. Object is cached after first parsing, so non-deterministic subcons will be affected. Works only with fixed size subcon.

Parameters **subcon** – the subcon to read/write on demand, must be fixed size

Example:

```
>>> OnDemand(Byte).parse(b"\xff")
<function OnDemand._parse.<locals>.<lambda> at 0x7fdc241cfc80>
>>> _()
255
>>> OnDemand(Byte).build(16)
b'\x10'
```

```
Can also re-build from the lambda returned at parsing.

>>> OnDemand(Byte).parse(b"\xff")
<function OnDemand._parse.<locals>.<lambda> at 0x7fcbd9855f28>
>>> OnDemand(Byte).build(_)
b'\xff'
```

`construct.core.`**`OnDemandPointer`**`(`*offset*, *subcon*`)`

An on-demand pointer. Is both lazy and jumps to a position before reading.

See also:

Base *`OnDemand()`* and *`Pointer()`* construct.

Parameters

- **`offset`** – an int or a context function that returns absolute stream position, where the construction would take place, can return negative integer as position from the end backwards
- **`subcon`** – the subcon that will be parsed or built at the *offset* stream position

Example:

```
>>> OnDemandPointer(lambda ctx: 2, Byte).parse(b"\x01\x02\x03garbage")
<function OnDemand._parse.<locals>.effectuate at 0x7f6f011ad510>
>>> _()
3
```

`construct.core.`**`OneOf`**`(`*subcon*, *valids*`)`

Validates that the object is one of the listed values, both during parsing and building.

Parameters

- **`subcon`** – a construct to validate
- **`valids`** – a collection implementing *in*

Example:

```
>>> OneOf(Byte, [1,2,3]).parse(b"\x01")
1
>>> OneOf(Byte, [1,2,3]).parse(b"\x08")
construct.core.ValidationError: ('invalid object', 8)

>>> OneOf(Bytes(1), b"1234567890").parse(b"4")
b'4'
>>> OneOf(Bytes(1), b"1234567890").parse(b"?")
construct.core.ValidationError: ('invalid object', b'?')

>>> OneOf(Bytes(2), b"1234567890").parse(b"78")
b'78'
>>> OneOf(Bytes(2), b"1234567890").parse(b"19")
construct.core.ValidationError: ('invalid object', b'19')
```

`construct.core.`**`Optional`**`(`*subcon*`)`

Makes an optional construct, that tries to parse the subcon. If parsing fails, returns None. If building fails, writes nothing.

Note: sizeof returns subcon size, although no bytes could be consumed or produced. Just something to consider.

Parameters **subcon** – the subcon to optionally parse or build

Example:

```
>>> Optional(Int64ul).parse(b"1234")
>>> Optional(Int64ul).parse(b"12345678")
4050765991979987505

>>> Optional(Int64ul).build(1)
b'\x01\x00\x00\x00\x00\x00\x00\x00'
>>> Optional(Int64ul).build("1")
b''
```

class `construct.core.`**`Padded`**(*length*, *subcon*, *pattern='x00'*, *strict=False*)
Appends additional null bytes to achieve a fixed length.

Example:

```
>>> Padded(4, Byte).build(255)
b'\xff\x00\x00\x00'
>>> Padded(4, Byte).parse(_)
255
>>> Padded(4, Byte).sizeof()
4

>>> Padded(4, VarInt).build(1)
b'\x01\x00\x00\x00'
>>> Padded(4, VarInt).build(70000)
b'\xf0\xa2\x04\x00'
```

`construct.core.`**`Padding`**(*length*, *pattern='\x00'*, *strict=False*)
A padding field that adds bytes when building, discards bytes when parsing.

Parameters

- **length** – length of the padding, an int or a function taking context and returning an int
- **pattern** – padding pattern as b-string character, default is b"x00" null character
- **strict** – whether to verify during parsing that the stream contains the pattern, raises an exception if actual padding differs from the pattern, default is False

Example:

```
>>> (Padding(4) >> Bytes(4)).parse(b"????abcd")
[None, b'abcd']
>>> (Padding(4) >> Bytes(4)).build(_)
b'\x00\x00\x00\x00abcd'
>>> (Padding(4) >> Bytes(4)).sizeof()
8

>>> Padding(4).build(None)
b'\x00\x00\x00\x00'
>>> Padding(4, strict=True).parse(b"****")
construct.core.PaddingError: expected b'\x00\x00\x00\x00', found b'****'
```

`construct.core.`**`PascalString`**(*lengthfield*, *encoding=None*)
A length-prefixed string.

`PascalString` is named after the string types of Pascal, which are length-prefixed. Lisp strings also follow this convention.

The length field will not appear in the same dict, when parsing. Only the string will be returned. When building, actual length is prepended before the encoded string. The length field can be variable length (such as VarInt). Stored length is in bytes, not characters.

Parameters

- **lengthfield** – a field used to parse and build the length
- **encoding** – encoding (eg. "utf8") or None for bytes

Example:

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

class `construct.core.`**`Peek`**(*subcon*)

Peeks at the stream. Parses without changing the stream position. If the end of the stream is reached when peeking, returns None. Sizeof returns 0 by design because build does not put anything into the stream. Building is no-op.

See also:

The *`Union()`* class.

Parameters **subcon** – the subcon to peek at

Example:

```
>>> Sequence(Peek(Byte), Peek(Int16ub)).parse(b"\x01\x02")
[1, 258]
>>> Sequence(Peek(Byte), Peek(Int16ub)).sizeof()
0
```

class `construct.core.`**`Pointer`**(*offset*, *subcon*)

Changes the stream position to a given offset, where the construction should take place, and restores the stream position when finished.

See also:

Analog *`OnDemandPointer()`* field, which also seeks to a given offset.

Parameters

- **offset** – an int or a function that takes context and returns absolute stream position, where the construction would take place, can return negative integer as position from the end backwards
- **subcon** – the subcon to use at the offset

Example:

```
>>> Pointer(8, Bytes(1)).parse(b"abcdefghijkl")
b'i'
>>> Pointer(8, Bytes(1)).build(b"x")
b'\x00\x00\x00\x00\x00\x00\x00\x00x'
>>> Pointer(8, Bytes(1)).sizeof()
0
```

class `construct.core.`**`Prefixed`**(*lengthfield*, *subcon*, *includelength=False*)

Parses the length field. Then reads that amount of bytes and parses the subcon using only those bytes. Constructs that consume entire remaining stream are constrained to consuming only the specified amount of bytes. When building, data is prefixed by its length. Optionally, length field can include its own size.

See also:

The `VarInt` encoding should be preferred over `Byte` and fixed size fields. VarInt is more compact and does never overflow.

Parameters

- **`lengthfield`** – a subcon used for storing the length
- **`subcon`** – the subcon used for storing the value
- **`includelength`** – optional, whether length field should include own size

Example:

```
>>> Prefixed(VarInt, GreedyBytes).parse(b"\x05hello?????")
b'hello'

>>>> Prefixed(VarInt, Byte[:]).parse(b"\x03\x01\x02\x03?????")
[1, 2, 3]
```

`construct.core.`**`PrefixedArray`**(*lengthfield*, *subcon*)

An array prefixed by a length field (as opposed to prefixed by byte count, see *`Prefixed()`*).

Parameters

- **`lengthfield`** – field parsing and building an integer
- **`subcon`** – subcon to process individual elements

Example:

```
>>> PrefixedArray(Byte, Byte).build(range(5))
b'\x05\x00\x01\x02\x03\x04'
>>> PrefixedArray(Byte, Byte).parse(_)
[0, 1, 2, 3, 4]
```

class `construct.core.`**`Range`**(*min*, *max*, *subcon*)

A homogenous array of elements. The array will iterate through between `min` to `max` times. If an exception occurs (EOF, validation error), the repeater exits cleanly. If less than `min` units have been successfully parsed, a RangeError is raised.

See also:

Analog *`GreedyRange()`* that parses until end of stream.

Note: This object requires a seekable stream for parsing.

Parameters

- **`min`** – the minimal count
- **`max`** – the maximal count
- **`subcon`** – the subcon to process individual elements

Example:

```
>>> Range(3, 5, Byte).build([1,2,3,4])
b'\x01\x02\x03\x04'
>>> Range(3, 5, Byte).parse(_)
[1, 2, 3, 4]

>>> Range(3, 5, Byte).build([1,2])
construct.core.RangeError: expected from 3 to 5 elements, found 2
>>> Range(3, 5, Byte).build([1,2,3,4,5,6])
construct.core.RangeError: expected from 3 to 5 elements, found 6
```

class `construct.core.`**`RawCopy`**(*subcon*)

Returns a dict containing both parsed subcon, the raw bytes that were consumed by it, starting and ending offset in the stream, and the amount of bytes. Builds either from raw bytes or a value used by subcon.

Context does contain a dict with data (if built from raw bytes) or with both (if built from value or parsed).

Example:

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

class `construct.core.`**`Rebuffered`**(*subcon*, *tailcutoff=None*)

Caches bytes from the underlying stream, so it becomes seekable and tellable. Also makes the stream blocking, in case it came from a socket or a pipe. Optionally, stream can forget bytes that went a certain amount of bytes beyond the current offset, allowing only a limited seeking capability while allowing to process an endless stream.

Warning: Experimental implementation. May not be mature enough.

Parameters

- **`subcon`** – the subcon which will operate on the buffered stream
- **`tailcutoff`** – optional, amount of bytes kept in buffer, by default buffers everything

Example:

```
Rebuffered(RepeatUntil(lambda obj,ctx: ?,Byte), tailcutoff=1024).parse_
↪stream(endless_nonblocking_stream)
```

class `construct.core.`**`Rebuild`**(*subcon*, *func*)

Parses the field like normal, but computes the value for building from a function. Useful for length and count fields when Prefixed and PrefixedArray cannot be used.

Example:

```
>>> st = Struct(
...     "count" / Rebuild(Byte, len_(this.items)),
...     "items" / Byte[this.count],
... )
```

```
>>> st.build(dict(items=[1,2,3]))
b'\x03\x01\x02\x03'
```

class `construct.core.`**`Renamed`**(*newname*, *subcon*)

Renames an existing construct. This creates a wrapper so underlying subcon retains it's original name, which in general means just a None. Can be used to give same construct few different names. Used internally by / operator.

Also this wrapper is responsible for building a path (a chain of names) that gets attached to error message when parsing, building, or sizeof fails. A field that is not named does not appear on the path.

Parameters

- **`newname`** – the new name
- **`subcon`** – the subcon to rename

Example:

```
>>> "name" / Int32ul
<Renamed: name>
>>> Renamed("name", Int32ul)
<Renamed: name>
```

class `construct.core.`**`RepeatUntil`**(*predicate*, *subcon*)

An array that repeats until the predicate indicates it to stop. Note that the last element (which caused the repeat to exit) is included in the return value.

Parameters

- **`predicate`** – a predicate function that takes (obj, list, context) and returns True to break or False to continue
- **`subcon`** – the subcon used to parse and build each element

Example:

```
>>> RepeatUntil(lambda x,lst,ctx: x>7, Byte).build(range(20))
b'\x00\x01\x02\x03\x04\x05\x06\x07\x08'
>>> RepeatUntil(lambda x,lst,ctx: x>7, Byte).parse(b"\x01\xff\x02")
[1, 255]
>>> RepeatUntil(lambda x,lst,ctx: lst[-2:]==[0,0], Byte).parse(b"\x01\x00\x00\xff
↪")
[1, 0, 0]
```

class `construct.core.`**`Restreamed`**(*subcon*, *encoder*, *encoderunit*, *decoder*, *decoderunit*, *decoderunitname*, *sizecomputer*)

Transforms bytes between the underlying stream and the subcon.

When the parsing or building is done, the wrapper stream is closed. If read buffer or write buffer is not empty, error is raised.

See also:

Both *`Bitwise()`* and *`Bytewise()`* are implemented using Restreamed.

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 after parsing/building. Also do NOT use pointers inside.

Parameters

- **subcon** – the subcon which will operate on the buffer
- **encoder** – a function that takes a b-string and returns a b-string (used when building)
- **encoderunit** – ratio as int, encoder takes that many bytes at once
- **decoder** – a function that takes a b-string and returns a b-string (used when parsing)
- **decoderunit** – ratio as int, decoder takes that many bytes at once
- **decoderunitname** – English string that describes the units (plural) returned by the decoder. Used for error messages.
- **sizecomputer** – a function that computes amount of bytes outputed by some bytes

Example:

```
Bitwise  <--> Restreamed(subcon, bits2bytes, 8, bytes2bits, 1, lambda n: n//8)
Bytewise <--> Restreamed(subcon, bytes2bits, 1, bits2bytes, 8, lambda n: n*8)
```

class `construct.core.`**`Seek`**`(`*at*, *whence=0*`)`

Sets a new stream position when parsing or building. Seeks are useful when many other fields follow the jump. Pointer works when there is only one field to look at, but when there is more to be done, Seek may come useful.

See also:

Analog *`Pointer()`* wrapper that has same side effect but also processed a subcon.

Parameters

- **at** – where to jump to, can be an int or a context lambda
- **whence** – is the offset from beginning (0) or from current position (1) or from ending (2), can be an int or a context lambda, default is 0

Example:

```
>>> (Seek(5) >> Byte).parse(b"01234x")
[5, 120]
>>> (Bytes(10) >> Seek(5) >> Byte).build([b"0123456789", None, 255])
b'01234\xff6789'
```

class `construct.core.`**`Select`**`(`**subcons*, ***kw*`)`

Selects the first matching subconstruct. It will literally try each of the subconstructs, until one matches.

Parameters

- **subcons** – the subcons to try (order sensitive)
- **includename** – indicates whether to include the name of the selected subcon in the return value of parsing, default is false

Example:

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

Note that this syntax works ONLY on python 3.6 due to ordered keyword arguments:
>>> Select(num=Int32ub, text=CString(encoding="utf8"))
```

class `construct.core.`**`Sequence`**`(`**subcons*, ***kw*`)`

A sequence of unnamed constructs. The elements are parsed and built in the order they are defined.

See also:

Can be nested easily, and embedded using *`Embedded()`* wrapper that merges entries into parent's entries.

Parameters **`subcons`** – a sequence of subconstructs that make up this sequence

Example:

```
>>> (Byte >> Byte).build([1, 2])
b'\x01\x02'
>>> (Byte >> Byte).parse(_)
[1, 2]
>>> (Byte >> Byte).sizeof()
2

>>> Sequence(Byte, CString(), Float32b).build([255, b"hello", 123])
b'\xffhello\x00B\xf6\x00\x00'
>>> Sequence(Byte, CString(), Float32b).parse(_)
[255, b'hello', 123.0]
```

class `construct.core.`**`Slicing`**`(`*subcon*, *count*, *start*, *stop*, *step=1*, *empty=None*`)`

Adapter for slicing a list (getting a slice from that list). Works with Range and Sequence and their lazy equivalents.

Parameters

- **`subcon`** – the subcon to slice
- **`count`** – expected number of elements, needed during building
- **`start`** – start index (or None for entire list)
- **`stop`** – stop index (or None for up-to-end)
- **`step`** – step (or 1 for every element)
- **`empty`** – value to fill the list with during building

Example:

```
???
```

class `construct.core.`**`StopIf`**`(`*condfunc*`)`

Checks for a condition, and stops a Struct Sequence Range from parsing or building.

Warning: May break sizeof methods. Unsure.

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)))
```

`construct.core.`**`String`**(*length*, *encoding=None*, *padchar='\x00'*, *paddir='right'*, *trimdir='right'*)

A configurable, fixed-length or variable-length string field.

When parsing, the byte string is stripped of pad character (as specified) from the direction (as specified) then decoded (as specified). Length is a constant integer or a function of the context. When building, the string is encoded (as specified) then padded (as specified) from the direction (as specified) or trimmed as bytes (as specified).

The padding character and direction must be specified for padding to work. The trim direction must be specified for trimming to work.

Parameters

- **`length`** – length in bytes (not unicode characters), as int or context function
- **`encoding`** – encoding (e.g. "utf8") or None for bytes
- **`padchar`** – b-string character to pad out strings (by default b"x00")
- **`paddir`** – direction to pad out strings (one of: right left both)
- **`trimdir`** – direction to trim strings (one of: right left)

Example:

```
>>> String(10).build(b"hello")
b'hello\x00\x00\x00\x00\x00'
>>> String(10).parse(_)
b'hello'
>>> String(10).sizeof()
10

>>> String(10, encoding="utf8").build("")
b'\xd0\x90\xd1\x84\xd0\xbe\xd0\xbd\x00\x00'
>>> String(10, encoding="utf8").parse(_)
''

>>> String(10, padchar=b"XYZ", paddir="center").build(b"abc")
b'XXXabcXXXX'
>>> String(10, padchar=b"XYZ", paddir="center").parse(b"XYZabcXYZY")
b'abc'

>>> String(10, trimdir="right").build(b"12345678901234567890")
b'1234567890'
```

class `construct.core.`**`StringEncoded`**(*subcon*, *encoding*)

Used internally.

class `construct.core.`**`StringPaddedTrimmed`**(*length*, *subcon*, *padchar='x00'*, *paddir='right'*, *trimdir='right'*)

Used internally.

class `construct.core.`**`Struct`**(**subcons*, ***kw*)

A sequence of usually named constructs, similar to structs in C. The elements are parsed and built in the order they are defined.

Some fields do not need to be named, since they are built from None anyway. See Const Padding Pass Terminated.

See also:

Can be nested easily, and embedded using *`Embedded()`* wrapper that merges members into parent's members.

Parameters **subcons** – a sequence of subconstructs that make up this structure

Example:

```
>>> Struct("a"/Int8ul, "data"/Bytes(2), "data2"/Bytes(this.a)).parse(b"\x01abc")
Container(a=1)(data=b'ab')(data2=b'c')
>>> Struct("a"/Int8ul, "data"/Bytes(2), "data2"/Bytes(this.a)).build(_)
b'\x01abc'
>>> Struct("a"/Int8ul, "data"/Bytes(2), "data2"/Bytes(this.a)).build(dict(a=5,␣
↪data=b"??", data2=b"hello"))
b'\x05??hello'

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

Note that this syntax works ONLY on python 3.6 due to ordered keyword arguments:
>>> 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, so it sizeof.

Subconstructs wrap an inner Construct, inheriting its name and flags.

Parameters **subcon** – the construct to wrap

class `construct.core.`**`Switch`**(*keyfunc*, *cases*, *default=<NoDefault: None>*, *includekey=False*)

A conditional branch. Switch will choose the case to follow based on the return value of keyfunc. If no case is matched, and no default value is given, SwitchError will be raised.

Warning: You can use Embedded(Switch(...)) but not Switch(Embedded(...)). Sames applies to If and IfThenElse macros.

Parameters

- **keyfunc** – a context function that returns a key which will choose a case, or a constant
- **cases** – a dictionary mapping keys to subcons
- **default** – a default field to use when the key is not found in the cases. if not supplied, an exception will be raised when the key is not found. Pass can be used for do-nothing
- **includekey** – whether to include the key in the return value of parsing, defualt is False

Example:

```
>>> Switch(this.n, { 1:Byte, 2:Int32ub }).build(5, dict(n=1))
b'\x05'
>>> Switch(this.n, { 1:Byte, 2:Int32ub }).build(5, dict(n=2))
b'\x00\x00\x00\x05'
```

class `construct.core.`**`SymmetricAdapter`**(*subcon*)

Abstract adapter parent class.

Needs to implement `_decode()` only. Encoding is done by same method.

Parameters **subcon** – the construct to wrap

construct.core.**SymmetricMapping**(*subcon*, *mapping*, *default=NotImplemented*)
Defines a symmetrical mapping, same mapping is used on parsing and building.

See also:

Based on *Mapping()*.

Parameters

- **subcon** – the subcon to map
- **encoding** – the mapping as a dict
- **decdefault** – the default return value when object is not found in the mapping, if no object is given an exception is raised, if Pass is used, the unmapped object will be passed as-is

Example:

```
???
```

class construct.core.**Union**(*parsefrom*, **subcons*, ***kw*)
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 if None, 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 the *parsefrom* parameter then stream will be left back at the starting offset. Many users fail to use this class properly.

Parameters

- **parsefrom** – how to leave stream after parsing, can be integer index or string name selecting a subcon, None (leaves stream at initial offset, the default), a context lambda returning either of previously mentioned
- **subcons** – subconstructs (order and name sensitive)

Example:

```
>>> Union(0, "raw"/Bytes(8), "ints"/Int32ub[2], "shorts"/Int16ub[4], "chars"/
↪Byte[8]).parse(b"12345678")
Container(raw=b'12345678')(ints=[825373492, 892745528])(shorts=[12594, 13108,␣
↪13622, 14136])(chars=[49, 50, 51, 52, 53, 54, 55, 56])

>>> Union(0, "raw"/Bytes(8), "ints"/Int32ub[2], "shorts"/Int16ub[4], "chars"/
↪Byte[8]).build(dict(chars=range(8)))
b'\x00\x01\x02\x03\x04\x05\x06\x07'

Note that this syntax works ONLY on python 3.6 due to ordered keyword arguments:
>>> Union(0, raw=Bytes(8), ints=Int32ub[2], shorts=Int16ub[4], chars=Byte[8])
>>> Union(0, raw=Bytes(8), ints=Int32ub[2], shorts=Int16ub[4], chars=Byte[8])
```

class `construct.core.`**`Validator`**(*subcon*)

Abstract class: validates a condition on the encoded/decoded object.

Needs to implement `_validate()` that returns bool.

Parameters **`subcon`** – the subcon to validate

`construct.core.`**`setglobalstringencoding`**(*encoding*)

Sets the encoding globally for all String PascalString CString GreedyString instances.

Parameters **`encoding`** – a string like "utf8" etc or None, which means working with bytes

`construct.lib` – entire module

class `construct.lib.`**`Container`**(**args*, ***kw*)

Generic ordered dictionary that allows both key and attribute access, and preserve key order by insertion. Also it uses __call__ method to chain add keys, because **kw does not preserve order.

Struct and Sequence, and few others parsers returns a container, since their members have order so do keys.

Example:

```
Container([ ("name","anonymous"), ("age",21) ])

Container(name="anonymous")(age=21)

# Note that this syntax does NOT work before python 3.6 due to unordered keyword␣
↪arguments:
Container(name="anonymous", age=21)

Container(container2)
```

`pop`(*key*, **default*)

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.

class `construct.lib.`**`FlagsContainer`**(**args*, ***kw*)

Container made to represent a FlagsEnum, only equality skips order. Provides pretty-printing for flags. Only set flags are displayed.

class `construct.lib.`**`ListContainer`**

A generic container for lists. Provides pretty-printing.

class `construct.lib.`**`LazyContainer`**(*keysbackend*, *offsetmap*, *cached*, *stream*, *addoffset*, *context*)

Lazy equivalent to Container. Works the same but parses subcons on first access whenever possible.

class `construct.lib.`**`LazyRangeContainer`**(*subcon*, *subsize*, *count*, *stream*, *addoffset*, *context*)

Lazy equivalent to ListContainer. Works the same but parses subcons on first access whenever possible.

class `construct.lib.`**`LazySequenceContainer`**(*count*, *offsetmap*, *cached*, *stream*, *addoffset*, *context*)

Lazy equivalent to ListContainer. Works the same but parses subcons on first access whenever possible.

`construct.lib.`**`integer2bits`**(*number*, *width*)

Converts an integer into its binary representation in a b-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 b'x00' or b'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 b-string into an integer. This is reverse to *bytes2integer*.

Examples:

```
>>> integer2bytes(19,4)
'\x00\x00\x00\x13'
```

`construct.lib.`**`bits2integer`**(*data*, *signed=False*)

Converts a b-string into an integer. Both b'0' and b'x00' are considered zero, and both b'1' and b'x01' are considered one. 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
>>> bits2integer(b"10011")
19
```

`construct.lib.`**`bytes2integer`**(*data*, *signed=False*)

Converts a b-string into an integer. This is reverse to *integer2bytes*.

Examples:

```
>>> bytes2integer(b'\x00\x00\x00\x13')
19
```

`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.`**`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.`**`swapbytes`**(*data*, *linesize=8*)

Performs an endianness swap on a b-string.

Example:

```
>>> swapbytes(b'00011011', 2)
b'11100100'
>>> swapbytes(b'0000000011111111', 8)
b'1111111100000000'
```

class `construct.lib.`**`HexString`**(*data*, *linesize=16*)

Represents bytes that will be hex-dumped when parsing, and un-dumped when building.

See hexdump().

`construct.lib.`**`hexdump`**(*data*, *linesize*)

Turns bytes into a unicode string of the format:

>>>print(hexdump(b'0' * 100, 16)) 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

`construct.lib.`**`hexundump`**(*data*, *linesize*)

Reverse of `hexdump()`.

`construct.lib.`**`int2byte`**(*i*)

Converts int (0 through 255) into b'...' character.

`construct.lib.`**`byte2int`**(*s*)

Converts b'...' character into int (0 through 255).

`construct.lib.`**`str2bytes`**(*s*)

Converts '...' str into b'...' bytes. On PY2 they are equivalent.

`construct.lib.`**`bytes2str`**(*b*)

Converts b'...' bytes into str. On PY2 they are equivalent.

`construct.lib.`**`str2unicode`**(*b*)

Converts '...' str into u'...' unicode string. On PY3 they are equivalent.

`construct.lib.`**`unicode2str`**(*s*)

Converts u'...' string into '...' str. On PY3 they are equivalent.

`construct.lib.`**`iteratebytes`**(*s*)

Iterates though b'...' string yielding characters as b'...' characters. On PY2 iter is the same.

`construct.lib.`**`iterateints`**(*s*)

Iterates though b'...' string yielding characters as ints. On PY3 iter is the same.

`construct.lib.`**`setglobalfullprinting`**(*enabled*)

Sets full printing for all Container instances. When enabled, Container str produces full content of bytes and strings, otherwise and by default, it produces truncated output.

Parameters **`enabled`** – bool to enable or disable full printing, or None to default

`construct.lib.`**`getglobalfullprinting`**()

Used internally.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index