Construct Documentation

Release 2.5

Tomer Filiba

October 17, 2016

Contents

1 Example **3**

2 Resources **5**

3 Requirements **7**

4 User Guide **9**
- 4.1 Introduction . . . 9
- 4.2 The Basics . . . 10
- 4.3 History . . . 16
- 4.4 Adapting . . . 18
- 4.5 Validating . . . 20
- 4.6 The Context . . . 21
- 4.7 Strings . . . 26
- 4.8 Miscellaneous . . . 28
- 4.9 Extending Construct . . . 32
- 4.10 Debugging Construct . . . 34

5 API Reference **37**
- 5.1 `construct.core` – Core data structures . . . 37
- 5.2 `construct.adapters` – Adapters . . . 47
- 5.3 `construct.macros` – Macros . . . 50
- 5.4 `construct.debug` – Debugging . . . 59

6 Indices and tables **61**

Python Module Index **63**

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, convert ("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 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
- 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 only what you require
- Pointers: jump from here to there in the data stream

CHAPTER 1

Example

A `PascalString` is a string prefixed by its length:

```
>>> from construct import *
>>>
>>> PascalString = Struct("PascalString",
...     UBInt8("length"),
...     Bytes("data", lambda ctx: ctx.length),
... )
>>>
>>> PascalString.parse("\x05helloXXX")
Container({'length': 5, 'data': 'hello'})
>>> PascalString.build(Container(length = 6, data = "foobar"))
'\x06foobar'
```

Instead of specifying the length manually, let's use an adapter:

```
>>> PascalString2 = ExprAdapter(PascalString,
...     encoder = lambda obj, ctx: Container(length = len(obj), data = obj),
...     decoder = lambda obj, ctx: obj.data
... )
>>> PascalString2.parse("\x05hello")
'hello'
>>> PascalString2.build("i'm a long string")
"\x11i'm a long string"
```

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

CHAPTER 2

Resources

Construct's homepage is http://construct.readthedocs.org, where you can find all kinds of docs and resources. The library itself is developed on github; please use github issues to report bugs, and github pull-requests to send in patches. For general discussion or questions, please use the new discussion group.

CHAPTER 3

Requirements

Construct should run on any Python 2.5-3.3 implementation.

Its only requirement is six, which is used to overcome the differences between Python 2 and 3.

CHAPTER 4

User Guide

4.1 Introduction

4.1.1 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**, or DSL, for specifying their data structures.

Binary

Construct operates on bytes, not strings, and is specialized for binary data. While Construct can consume normal text, it is best suited for binary formats.

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.

4.1.2 What is Construct good for?

Construct has been used to parse:

- Networking formats
- Binary file formats

- Filesystem layouts

And many other things!

4.1.3 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. While Construct does have a suite of special text-parsing structures, it was not designed to handle text and is not a good fit for those applications.

4.2 The Basics

4.2.1 Fields

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

Some examples of parsing:

```
>>> from construct import UBInt16, ULInt16
>>> UBInt16("foo").parse("\x01\x02")
258
>>> ULInt16("foo").parse("\x01\x02")
513
```

Some examples of building:

```
>>> from construct import UBInt16, SBInt16
>>> UBInt16("foo").build(31337)
'zi'
>>> SBInt16("foo").build(-31337)
'\x86\x97'
```

4.2.2 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. Note that if two or more fields of a Struct have the same name, the last field "wins"; that is, the last field's value will be the value returned from a parse.

```
>>> from construct import Struct, UBInt8, SLInt16, LFloat32
>>> c = Struct("foo",
...     UBInt8("a"),
...     SLInt16("b"),
...     LFloat32("c"),
... )
>>> c
<Struct('foo')>
>>> c.parse("\x07\x00\x01\x00\x00\x00\x01")
Container(a = 7, b = 256, c = 2.350988701644575e-038)
```

Containers

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

```
>>> x = c.parse("\x07\x00\x01\x00\x00\x00\x01")
>>> x
Container(a = 7, b = 256, c = 2.350988701644575e-038)
>>> x.a
7
>>> x.b
256
>>> print x
Container:
    a = 7
    b = 256
    c = 2.350988701644575e-038
```

Building

And here is how we build Structs:

```
>>> # Rebuild the parsed object.
>>> c.build(x)
'\x07\x00\x01\x00\x00\x00\x01'
```

```
>>> # Mutate the parsed object and build...
>>> x.b = 5000
>>> c.build(x)
'\x07\x88\x13\x00\x00\x00\x01'
```

```
>>> # ...Or, we can create a new container.
>>> c.build(Container(a = 9, b = 1234, c = 56.78))
'\t\xd2\x04\xb8\x1ecB'
```

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

```
>>> class Foo(object): pass
...
>>> f = Foo()
>>> f.a = 1
>>> f.b = 2
>>> f.c = 3
>>> c.build(f)
'\x01\x02\x00\x00\x00@@'
```

Nested

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

```
>>> c = Struct("foo",
...     UBInt8("a"),
...     UBInt16("b"),
...     Struct("bar",
```

```
...         UBInt8("a"),
...         UBInt16("b"),
...     )
... )
>>> x = c.parse("ABBabb")
>>> x
Container(a = 65, b = 16962, bar = Container(a = 97, b = 25186))
>>> print x
Container:
    a = 65
    b = 16962
    bar = Container:
        a = 97
        b = 25186
>>> x.a
65
>>> x.bar
Container(a = 97, b = 25186)
>>> x.bar.b
25186
```

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

Embedding

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.

```
>>> foo = Struct("foo",
...     UBInt8("a"),
...     UBInt8("b"),
... )
>>> bar = Struct("bar",
...     foo, # This Struct is not embedded.
...     UBInt8("c"),
...     UBInt8("d"),
... )
>>> bar2= Struct("bar",
...     Embed(foo), # This Struct is embedded.
...     UBInt8("c"),
...     UBInt8("d"),
... )
>>> bar.parse("abcd")
Container(c = 99, d = 100, foo = Container(a = 97, b = 98))
>>> bar2.parse("abcd")
Container(a = 97, b = 98, c = 99, d = 100)
```

See also:

The *Embedded()* macro.

4.2.3 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.

Parsing

```
>>> c = Sequence("foo",
...     UBInt8("a"),
...     UBInt16("b"),
... )
>>> c
<Sequence('foo')>
>>> c.parse("abb")
[97, 25186]
```

Building

```
>>> c.build([1,2])
'\x01\x00\x02'
```

Nested

```
>>> c = Sequence("foo",
...     UBInt8("a"),
...     UBInt16("b"),
...     Sequence("bar",
...         UBInt8("a"),
...         UBInt16("b"),
...     )
... )
>>> c.parse("ABBabb")
[65, 16962, [97, 25186]]
```

Embedded

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

```
>>> foo = Sequence("foo",
...     UBInt8("a"),
...     UBInt8("b"),
... )
>>> bar = Sequence("bar",
...     foo,                    # <-- unembedded
...     UBInt8("c"),
...     UBInt8("d"),
... )
>>> bar2 = Sequence("bar",
...     Embed(foo),             # <-- embedded
...     UBInt8("c"),
...     UBInt8("d"),
... )
>>> bar.parse("abcd")
[[97, 98], 99, 100]
```

```
>>> bar2.parse("abcd")
[97, 98, 99, 100]
```

4.2.4 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. Meta-repeaters will be covered in the meta-constructs tutorial.

We have four kinds of static repeaters. In fact, for those of you who wish to go under the hood, two of these repeaters are actually wrappers around Range.

construct.**Range**(*mincount*, *maxcout*, *subcon*)

A range-array. The subcon will iterate between `mincount` to `maxcount` times. If less than `mincount` elements are found, raises RangeError.

See also:

The *GreedyRange()* and *OptionalGreedyRange()* macros.

The general-case repeater. Repeats the given unit for at least `mincount` times, and up to `maxcount` times. If an exception occurs (EOF, validation error), the repeater exits. If less than `mincount` units have been successfully parsed, a RangeError is raised.

Note: This object requires a seekable stream for parsing.

Parameters

- **mincount** – the minimal count
- **maxcount** – the maximal count
- **subcon** – the subcon to repeat

Example:

```
>>> c = Range(3, 7, UBInt8("foo"))
>>> c.parse("\x01\x02")
Traceback (most recent call last):
  ...
construct.core.RangeError: expected 3..7, found 2
>>> c.parse("\x01\x02\x03")
[1, 2, 3]
>>> c.parse("\x01\x02\x03\x04\x05\x06")
[1, 2, 3, 4, 5, 6]
>>> c.parse("\x01\x02\x03\x04\x05\x06\x07")
[1, 2, 3, 4, 5, 6, 7]
>>> c.parse("\x01\x02\x03\x04\x05\x06\x07\x08\x09")
[1, 2, 3, 4, 5, 6, 7]
>>> c.build([1,2])
Traceback (most recent call last):
  ...
construct.core.RangeError: expected 3..7, found 2
>>> c.build([1,2,3,4])
'\x01\x02\x03\x04'
>>> c.build([1,2,3,4,5,6,7,8])
Traceback (most recent call last):
```

```
    ...
construct.core.RangeError: expected 3..7, found 8
```

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

Repeats the given unit a fixed number of times.

Parameters

- **count** – number of times to repeat
- **subcon** – construct to repeat

Example:

```
>>> c = Array(4, UBInt8("foo"))
>>> c.parse("\x01\x02\x03\x04")
[1, 2, 3, 4]
>>> c.parse("\x01\x02\x03\x04\x05\x06")
[1, 2, 3, 4]
>>> c.build([5,6,7,8])
'\x05\x06\x07\x08'
>>> c.build([5,6,7,8,9])
Traceback (most recent call last):
  ...
construct.core.RangeError: expected 4..4, found 5
```

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

Repeats the given unit one or more times.

Parameters **subcon** – construct to repeat

Example:

```
>>> from construct import GreedyRange, UBInt8
>>> c = GreedyRange(UBInt8("foo"))
>>> c.parse("\x01")
[1]
>>> c.parse("\x01\x02\x03")
[1, 2, 3]
>>> c.parse("\x01\x02\x03\x04\x05\x06")
[1, 2, 3, 4, 5, 6]
>>> c.parse("")
Traceback (most recent call last):
  ...
construct.core.RangeError: expected 1..2147483647, found 0
>>> c.build([1,2])
'\x01\x02'
>>> c.build([])
Traceback (most recent call last):
  ...
construct.core.RangeError: expected 1..2147483647, found 0
```

construct.**OptionalGreedyRange**(*subcon*)

Repeats the given unit zero or more times. This repeater can't fail, as it accepts lists of any length.

Parameters **subcon** – construct to repeat

Example:

```
>>> from construct import OptionalGreedyRange, UBInt8
>>> c = OptionalGreedyRange(UBInt8("foo"))
>>> c.parse("")
```

```
    []
    >>> c.parse("\x01\x02")
    [1, 2]
    >>> c.build([])
    ''
    >>> c.build([1,2])
    '\x01\x02'
```

Nesting

As with all constructs, Repeaters can be nested too. Here's an example:

```
>>> c = Array(5, Array(2, UBInt8("foo")))
>>> c.parse("aabbccddee")
[[97, 97], [98, 98], [99, 99], [100, 100], [101, 101]]
```

4.3 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/bits-to-bytes coversions
- Can rely on python's built in struct module for numeric packing/unpacking (faster, tested)
- Can directly parse-from/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: BitFields handle parsing/building bit fields, and BitStructs handle to/from conversions.

4.3.1 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 operate 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, see the code snippet in the BitField section.

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

Important notes

- Non-nestable - BitStructs are not nestable/stackable; writing something like `BitStruct("foo", BitStruct("bar", Octet("spam")))` will not work. You can use regular Structs inside BitStructs.
- BitStructs are embeddable.

  See also:

  The *`EmbeddedBitStruct()`* macro.
- Byte-Aligned - The total size of the elements of a BitStruct must be a multiple of 8 (due to alignment issues)
- Pointers and OnDemand - Do not place Pointers or OnDemands inside BitStructs, since it uses an internal stream, so external stream offsets will turn out wrong.

4.3.2 BitField

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

BitFields, as the name suggests, are fields that operate on raw, unaligned bits, and therefore must be enclosed in a BitStruct. Using them is very similar to all normal fields: they take a name and a length (in bits).

Parameters

- **`name`** – name of the field
- **`length`** – number of bits in the field, or a function that takes the context as its argument and returns the length
- **`swapped`** – whether the value is byte-swapped
- **`signed`** – whether the value is signed
- **`bytesize`** – number of bits per byte, for byte-swapping

Example:

```
>>> foo = BitStruct("foo",
...     BitField("a", 3),
...     Flag("b"),
...     Padding(3),
...     Nibble("c"),
...     BitField("d", 5),
... )
>>> foo.parse("\xe1\x1f")
Container(a = 7, b = False, c = 8, d = 31)
>>> foo = BitStruct("foo",
...     BitField("a", 3),
...     Flag("b"),
...     Padding(3),
...     Nibble("c"),
...     Struct("bar",
...             Nibble("d"),
...             Bit("e"),
...     )
... )
>>> foo.parse("\xe1\x1f")
Container(a = 7, b = False, bar = Container(d = 15, e = 1), c = 8)
```

Convenience wrappers for BitField

Bit A single bit

Nibble A sequence of 4 bits (half a byte)

Octet An sequence of 8 bits (byte)

4.3.3 The Bit/Byte Duality

Most simple fields (such as Flag, Padding, Terminator, 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:

```
>>> c1 = Struct("foo",
...     Padding(2),
...     Flag("myflag"),
...     Padding(5),
... )
>>>
>>> c1.parse("\x00\x00\x01\x00\x00\x00\x00\x00")
Container(myflag = True)
```

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

```
>>> c2 = BitStruct("foo",
...     Padding(2),
...     Flag("myflag"),
...     Padding(5),
... )
>>>
>>> c2.parse("\x20")
Container(myflag = True)
>>>
```

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.

4.4 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 "".join(chr(int(b)) for b in obj.split("."))
...     def _decode(self, obj, context):
...         return ".".join(str(ord(b)) for b in obj)
...
```

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 the object!

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:

```
>>> IpAddressAdapter(Bytes("foo", 4)).parse("\x01\x02\x03\x04")
'1.2.3.4'
>>> IpAddressAdapter(Bytes("foo", 4)).build("192.168.2.3")
'\xc0\xa8\x02\x03'
```

We can also use macro functions, to bind an adapter to a construct, instead of having to do so manually every time:

```
>>> def IpAddress(name):
...     return IpAddressAdapter(Bytes(name, 4))
...
>>> IpAddress("foo").build("10.0.0.1")
'\n\x00\x00\x01'
```

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 `UBInt8`, `SLInt32`, or `LFloat64`, etc., as long as the underlying construct returns an object we can map. Moreover, we can stack several adapters on top of one another, to created a nested adapter.

4.4.1 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 `MappingAdapter`, which provides mapping of values to other values (not necessarily names to numbers).

```
>>> c = Enum(Byte("protocol"),
...     TCP = 6,
...     UDP = 17,
... )
>>> c
<MappingAdapter('protocol')>
```

```
>>> # parsing
>>> c.parse("\x06")
'TCP'
>>> c.parse("\x11")
'UDP'
>>> c.parse("\x12")
Traceback (most recent call last):
  .
  .
construct.adapters.MappingAdapterError: undefined mapping for 18
```

```
>>> # building
>>> c.build("TCP")
'\x06'
>>> c.build("UDP")
```

```
'\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_` (a single uderscore on each side). If we don't supply a default value, an exception is raised (as we saw in the previous snippet).

```
>>> c = Enum(Byte("protocol"),
...     TCP = 6,
...     UDP = 17,
...     _default_ = "blah"
... )
>>> c.parse("\x11")
'UDP'
>>> c.parse("\x12") # no mapping for 18, so default to "blah"
'blah'
>>>
```

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("protocol"),
...     TCP = 6,
...     UDP = 17,
...     _default_ = Pass
... )
>>> c.parse("\x11")
'UDP'
>>> c.parse("\x12") # no mapping, passing through
18
>>> c.parse("\xff") # no mapping, passing through
255
```

When we wish to use the same enum multiple times, we will use a simple macro function. This keeps us conforming to the Don't Repeat Yourself principle:

```
>>> def ProtocolEnum(subcon):
...     return Enum(subcon,
...         ICMP = 1,
...         TCP = 6,
...         UDP = 17,
...     )
...
>>> ProtocolEnum(UBInt8("foo")).parse("\x06")
'TCP'
>>> ProtocolEnum(UBInt16("foo")).parse("\x00\x06")
'TCP'
>>>
```

4.5 Validating

Validating means making sure the parsed/built object meets a given condition. Validators simply raise an exception (`ValidatorError`) if the object is invalid. The two most common cases already exist as builtins.

Validators are usually used to make sure a "magic number" is found, the correct version of the protocol, a file signature is matched, etc. 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, etc.

class construct.**NoneOf**(*subcon*, *invalids*)
Validates that the object is none of the listed values.

Parameters

- **subcon** – object to validate
- **invalids** – a set of invalid values

Example:

```
>>> NoneOf(UBInt8("foo"), [4,5,6,7]).parse("\x08")
8
>>> NoneOf(UBInt8("foo"), [4,5,6,7]).parse("\x06")
Traceback (most recent call last):
    ...
construct.core.ValidationError: ('invalid object', 6)
```

class construct.**OneOf**(*subcon*, *valids*)
Validates that the object is one of the listed values.

Parameters

- **subcon** – object to validate
- **valids** – a set of valid values

Example:

```
>>> OneOf(UBInt8("foo"), [4,5,6,7]).parse("\x05")
5
>>> OneOf(UBInt8("foo"), [4,5,6,7]).parse("\x08")
Traceback (most recent call last):
    ...
construct.core.ValidationError: ('invalid object', 8)
>>>
>>> OneOf(UBInt8("foo"), [4,5,6,7]).build(5)
'\x05'
>>> OneOf(UBInt8("foo"), [4,5,6,7]).build(9)
Traceback (most recent call last):
    ...
construct.core.ValidationError: ('invalid object', 9)
```

4.6 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 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. 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):
...         print context
...
```

```
>>> foo = Struct("foo",
...     Byte("a"),
...     Byte("b"),
...     PrintContext("c"),
...     Struct("bar",
...         Byte("a"),
...         Byte("b"),
...         PrintContext("c"),
...     ),
...     PrintContext("d"),
... )
>>>
>>> foo.parse("\x01\x02\x03\x04")
{'_': {'a': 1, 'b': 2}}
{'_': {'a': 3, 'b': 4, '_': {'a': 1, 'c': None, 'b': 2}}}
{'_': {'a': 1, 'c': None, 'b': 2, 'bar': Container(a = 3, b = 4, c = None)}}
Container(a = 1, b = 2, bar = Container(a = 3, b = 4, c = None), c = None, d =
None)
>>>
```

As you can see, the context looks different in 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.

```
>>> foo = Struct("foo",
...     Byte("length"),
...     Field("data", lambda ctx: ctx.length * 2 + 1),   # <-- calculate
the length of the string
... )
>>>
>>> foo.parse("\x05abcdefghijkXXX")
Container(data = 'abcdefghijk', length = 5)
```

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

```
>>> foo = Struct("foo",
...     Byte("length"),
...     Field("data", lambda ctx: 7),
... )
>>>
>>> foo.parse("\x99abcdefg")
Container(data = 'abcdefg', length = 153)
```

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`.

```
>>> foo = Struct("foo",
...     Byte("length1"),
...     Struct("bar",
...         Byte("length2"),
...         Field("data", lambda ctx: ctx._.length1 + ctx.length2),
...     )
... )
>>>
```

```
>>> foo.parse("\x02\x03abcde")
Container(bar = Container(data = 'abcde', length2 = 3), length1 = 2)
```

construct.**Field**(*name*, *length*)

A field consisting of a specified number of bytes.

Parameters

- **name** – the name of the field
- **length** – the length of the field. the length can be either an integer (StaticField), or a function that takes the context as an argument and returns the length (MetaField)

4.6.1 Array

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

```
>>> foo = Struct("foo",
...     Byte("length"),
...     Array(lambda ctx: ctx.length, UBInt16("data")),
... )
>>>
>>> foo.parse("\x03\x00\x01\x00\x02\x00\x03")
Container(data = [1, 2, 3], length = 3)
```

4.6.2 RepeatUntil

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

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

```
>>> foo = RepeatUntil(lambda obj, ctx: obj == "\x00", Field("data", 1))
>>>
>>> foo.parse("abcdef\x00this is another string")
['a', 'b', 'c', 'd', 'e', 'f', '\x00']
>>>
>>> foo2 = StringAdapter(foo)
>>> foo2.parse("abcdef\x00this is another string")
'abcdef\x00'
```

4.6.3 Switch

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

```
>>> foo = Struct("foo",
...     Enum(Byte("type"),
...         INT1 = 1,
...         INT2 = 2,
...         INT4 = 3,
...         STRING = 4,
...     ),
...     Switch("data", lambda ctx: ctx.type,
```

```
...         {
    ...             "INT1" : UBInt8("spam"),
    ...             "INT2" : UBInt16("spam"),
    ...             "INT4" : UBInt32("spam"),
    ...             "STRING" : String("spam", 6),
    ...         }
    ...     )
... )
>>>
>>>
>>> foo.parse("\x01\x12")
Container(data = 18, type = 'INT1')
>>>
>>> foo.parse("\x02\x12\x34")
Container(data = 4660, type = 'INT2')
>>>
>>> foo.parse("\x03\x12\x34\x56\x78")
Container(data = 305419896, type = 'INT4')
>>>
>>> foo.parse("\x04abcdef")
Container(data = 'abcdef', type = 'STRING')
```

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.

```
>>> foo = Struct("foo",
...     Byte("type"),
...     Switch("data", lambda ctx: ctx.type,
...         {
    ...             1 : UBInt8("spam"),
    ...             2 : UBInt16("spam"),
    ...         },
    ...         default = UBInt8("spam")            # <-- sets the default
    construct
    ...     )
... )
>>>
>>> foo.parse("\x01\xff")
Container(data = 255, type = 1)
>>>
>>> foo.parse("\x02\xff\xff")
Container(data = 65535, type = 2)
>>>
>>> foo.parse("\x03\xff\xff")                    # <-- uses the default
construct
Container(data = 255, type = 3)
>>>
```

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.

```
>>> foo = Struct("foo",
...     Byte("type"),
...     Switch("data", lambda ctx: ctx.type,
...         {
    ...             1 : UBInt8("spam"),
    ...             2 : UBInt16("spam"),
    ...         },
    ...         default = Pass
```

```
    ...     )
... )
>>>
>>> foo.parse("\x01\xff")
Container(data = 255, type = 1)
>>>
>>> foo.parse("\x02\xff\xff")
Container(data = 65535, type = 2)
>>>
>>> foo.parse("\x03\xff\xff")
Container(data = None, type = 3)
```

4.6.4 Pointer

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

Note: Pointers are available only for seekable streams (in-memory and files). Sockets and pipes do not suppose seeking, so you'll have to first read the data from the stream, and parse it in-memory.

```
>>> foo = Struct("foo",
...     Pointer(lambda ctx: 4, Byte("data1")),    # <-- data1 is at (absolute)
position 4
...     Pointer(lambda ctx: 7, Byte("data2")),    # <-- data2 is at (absolute)
position 7
... )
>>>
>>> foo.parse("\x00\x00\x00\x00\x01\x00\x00\x02")
Container(data1 = 1, data2 = 2)
```

4.6.5 Anchor

Anchor is not really a meta construct, but it strongly coupled with Pointer, so I chose to list it here. Anchor simply returns the stream position at the moment it's invoked, so Pointers can "anchor" relative offsets to absolute stream positions using it. See the following example:

```
>>> foo = Struct("foo",
...     Byte("padding_length"),
...     Padding(lambda ctx: ctx.padding_length),
...     Byte("relative_offset"),
...     Anchor("absolute_position"),
...     Pointer(lambda ctx: ctx.absolute_position + ctx.relative_offset,
...         Byte("data")
...     ),
... )
>>>
>>> foo.parse("\x05\x00\x00\x00\x00\x00\x03\x00\x00\x00\xff")
Container(absolute_position = 7, data = 255, padding_length = 5,
relative_offset = 3)
```

4.6.6 OnDemand

OnDemand allows lazy construction, meaning the data is actually parsed (or built) only when it's requested (demanded). On-demand parsing 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 OnDemand is an OnDemandContainer – a special object that "remembers" the stream position where its data is found, and parses it when you access its .value property.

Note: Lazy construction is available only for seekable streams (in-memory and files). Sockets and pipes do not suppose seeking, so you'll have to first read the data from the stream, and parse it in-memory.

```
>>> foo = Struct("foo",
...     Byte("a"),
...     OnDemand(Bytes("bigdata", 20)),  # <-- this will be read only on
demand
...     Byte("b"),
... )
>>>
>>> x = foo.parse("\x01012345678901234567890\x02")
>>> x
Container(a = 1, b = 2, bigdata = OnDemandContainer(<unread>))
>>>
>>> x.bigdata
OnDemandContainer(<unread>)
>>> x.bigdata.has_value                        # <-- still unread
False
>>>
>>> x.bigdata.value                            # <-- demand the data
'01234567890123456789'
>>>
>>> x.bigdata.has_value                        # <-- already demanded
True
>>> x.bigdata
OnDemandContainer('01234567890123456789')
```

4.7 Strings

Strings in Construct work very much like strings in other languages.

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

A configurable, fixed-length string field.

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

Parameters

- **name** – name
- **length** – length, in bytes
- **encoding** – encoding (e.g. "utf8") or None for no encoding
- **padchar** – optional character to pad out strings
- **paddir** – direction to pad out strings; one of "right", "left", or "both"
- **trim** (*str*) – direction to trim strings; one of "right", "left"

Example:

```
>>> from construct import String
>>> String("foo", 5).parse("hello")
'hello'
>>>
>>> String("foo", 12, encoding = "utf8").parse("hello joh\xd4\x83n")
u'hello joh\u0503n'
>>>
>>> foo = String("foo", 10, padchar = "X", paddir = "right")
>>> foo.parse("helloXXXXX")
'hello'
>>> foo.build("hello")
'helloXXXXX'
```

construct.**PascalString**(*name*, *length_field=FormatField('length')*, *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 appear in the same Container as the PascalString, with the given name.

Parameters

- **name** – name
- **length_field** – a field which will store the length of the string
- **encoding** – encoding (e.g. "utf8") or None for no encoding

Example:

```
>>> foo = PascalString("foo")
>>> foo.parse("\x05hello")
'hello'
>>> foo.build("hello world")
'\x0bhello world'
>>>
>>> foo = PascalString("foo", length_field = UBInt16("length"))
>>> foo.parse("\x00\x05hello")
'hello'
>>> foo.build("hello")
'\x00\x05hello'
```

construct.**CString**(*name*, *terminators='\x00'*, *encoding=None*, *char_field=StaticField(None)*)

A string ending in a terminator.

CString is similar to the strings of C, C++, and other related programming languages.

By default, the terminator is the NULL byte (b''0x00'').

Parameters

- **name** – name
- **terminators** – sequence of valid terminators, in order of preference
- **encoding** – encoding (e.g. "utf8") or None for no encoding
- **char_field** – construct representing a single character

Example:

```
    >>> foo = CString("foo")
    >>> foo.parse(b"hello\x00")
    b'hello'
    >>> foo.build(b"hello")
    b'hello\x00'
    >>> foo = CString("foo", terminators = b"XYZ")
    >>> foo.parse(b"helloX")
    b'hello'
    >>> foo.parse(b"helloY")
    b'hello'
    >>> foo.parse(b"helloZ")
    b'hello'
    >>> foo.build(b"hello")
    b'helloX'
```

4.8 Miscellaneous

4.8.1 Conditional

Optional

Attempts to parse or build the subconstruct; if it fails, returns a default value. By default, the default value is `None`.

```
>>> foo = Optional(UBInt32("foo"))
>>> foo.parse("\x12\x34\x56\x78")
305419896
>>> print foo.parse("\x12\x34\x56")
None
>>>
>>> foo = Optional(UBInt32("foo"), default = 17)
>>> foo.parse("\x12\x34\x56\x78")
305419896
>>> foo.parse("\x12\x34\x56")
17
```

If

Parses or builds the subconstruct only if a certain condition is met. Otherwise, returns a default value. By default, the default value is `None`.

```
>>> foo = Struct("foo",
...     Flag("has_options"),
...     If(lambda ctx: ctx["has_options"],
...         Bytes("options", 5)
...     )
... )
>>>
>>> foo.parse("\x01hello")
Container(has_options = True, options = 'hello')
>>>
>>> foo.parse("\x00hello")
Container(has_options = False, options = None)
>>>
```

IfThenElse

Branches the construction path based on a given condition. If the condition is met, the `then_construct` is used; otherwise the `else_construct` is used.

```
>>> foo = Struct("foo",
...     Byte("a"),
...     IfThenElse("b", lambda ctx: ctx["a"] > 7,
...         UBInt32("foo"),
...         UBInt16("bar")
...     ),
... )
>>>
>>> foo.parse("\x09\xaa\xbb\xcc\xdd")     # <-- condition is met
Container(a = 9, b = 2864434397L)
>>> foo.parse("\x02\xaa\xbb")             # <-- condition is not met
Container(a = 2, b = 43707)
```

4.8.2 Alignment and Padding

Aligned

Aligns the subconstruct to a given modulus boundary (default is 4).

```
>>> foo = Aligned(UBInt8("foo"))
>>> foo.parse("\xff\x00\x00\x00")
255
>>> foo.build(255)
'\xff\x00\x00\x00'
```

AlignedStruct

Automatically aligns all the fields of the Struct to the modulus boundary.

```
>>> foo = AlignedStruct("foo",
...     Byte("a"),
...     Byte("b"),
... )
>>>
>>> foo.parse("\x01\x00\x00\x00\x02\x00\x00\x00")
Container(a = 1, b = 2)
>>> foo.build(Container(a=1,b=2))
'\x01\x00\x00\x00\x02\x00\x00\x00'
```

Padding

Padding is a sequence of bytes or bits that contains no data (its value is discarded), and is necessary only for padding, etc.

```
>>> foo = Struct("foo",
...     Byte("a"),
...     Padding(2),
...     Byte("b"),
... )
```

```
>>>
>>> foo.parse("\x01\x00\x00\x02")
Container(a = 1, b = 2)
```

4.8.3 Special Constructs

Rename

Renames a construct.

```
>>> foo = Struct("foo",
...     Rename("xxx", Byte("yyy")),
... )
>>>
>>> foo.parse("\x02")
Container(xxx = 2)
```

Alias

Creates an alias for an existing field of a Struct.

```
>>> foo = Struct("foo",
...     Byte("a"),
...     Alias("b", "a"),
... )
>>>
>>> foo.parse("\x03")
Container(a = 3, b = 3)
```

Value

Represents a computed value. Value does not read or write anything to the stream; it only returns its computed value as the result.

```
>>> foo = Struct("foo",
...     Byte("a"),
...     Value("b", lambda ctx: ctx["a"] + 7)
... )
>>>
>>> foo.parse("\x02")
Container(a = 2, b = 9)
```

Terminator

Asserts the end of the stream has been reached (so that no more trailing data is left unparsed).

Note: Terminator is a singleton object. Do not try to "instantiate" it (i.e., `Terminator()`).

```
>>> Terminator.parse("")
>>> Terminator.parse("x")
Traceback (most recent call last):
```

```
  .
  .
construct.extensions.TerminatorError: end of stream not reached
```

Pass

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

Note: Pass is a singleton object. Do not try to "instantiate" it (i.e., `Pass()`).

```
>>> print Pass.parse("xyz")
None
```

Const

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

```
>>> foo = Const(Bytes("magic", 6), "FOOBAR")
>>> foo.parse("FOOBAR")
'FOOBAR'
>>> foo.parse("FOOBAX")
Traceback (most recent call last):
  .
  .
construct.extensions.ConstError: expected 'FOOBAR', found 'FOOBAX'
>>>
```

Peek

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

Note: Works only with seekable streams (in-memory and files).

```
>>> foo = Struct("foo",
...     Byte("a"),
...     Peek(Byte("b")),
...     Byte("c"),
... )
>>> foo.parse("\x01\x02")
Container(a = 1, b = 2, c = 2)
```

Union

Treats the same data as multiple constructs (similar to C's union statement). When building, each subconstruct parses the same data (so you can "look" at the data in multiple views); when writing, the first subconstruct is used to build the final result.

Note: Works only with seekable streams (in-memory and files).

```
>>> foo = Union("foo",
...     UBInt32("a"),
...     UBInt16("b"),                                # <-- note that this field is
of a different size
...     Struct("c", UBInt16("high"), UBInt16("low")),
...     LFloat32("d"),
... )
>>>
>>> print foo.parse("\xaa\xbb\xcc\xdd")
Container:
    a = 2864434397L
    b = 43707
    c = Container:
        high = 43707
        low = 52445
    d = -1.8440714901698642e+018
>>>
>>> foo.build(Container(a = 0x11223344, b=0,c=Container(low=0, high=0),d=0)) #
<-- only "a" is used for building
'\x11"3D'
```

LazyBound

A lazy-bound construct; it 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).

```
>>> foo = Struct("foo",
...     Flag("has_next"),
...     If(lambda ctx: ctx["has_next"], LazyBound("next", lambda: foo)),
... )
>>>
>>> print foo.parse("\x01\x01\x01\x00")
Container:
    has_next = True
    next = Container:
        has_next = True
        next = Container:
            has_next = True
            next = Container:
                has_next = False
                next = None
>>>
```

4.9 Extending Construct

4.9.1 Adapters

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

In order to write custom adapters, implement `_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
```

4.9.2 Constructs

Generally speaking, you should not write constructs by yourself:

- It's a craft that requires skills and understanding of the internals of the library (which change over time).
- Adapters should really be all you need and are much more simpler to implement.
- To make things faster, try using psyco, or write your code in pyrex. 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... the reason there's no such construct yet is because I couldn't find an elegant way to do that (although Buffered or Union may be a good place to start).

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):
        # read from the stream (usually not directly)
        # return object
        pass

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

    def _sizeof(self, context):
        # return computed size, or raise SizeofError if not possible
        pass
```

Subconstructs

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

```
class MySubconstruct(Subconstruct):
    def _parse(self, stream, context):
        obj = self.subcon._parse(stream, context)
        # do something with obj
        # return object

    def _build(self, obj, stream, context):
        # do something with obj
        self.subcon._build(obj, stream, context)
        # no return value is necessary
```

4.10 Debugging Construct

4.10.1 Intro

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.

4.10.2 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.

```
>>> foo = Struct("foo",
...     UBInt8("bar"),
...     Probe(),
...     UBInt8("baz"),
... )
>>> foo.parse("spam spam spam spam bacon and eggs")
Probe <unnamed 1>
Container:
    stream_position = 1
    following_stream_data =
        0000   70 61 6d 20 73 70 61 6d 20 73 70 61 6d 20 73 70   pam spam spam
sp
        0010   61 6d 20 62 61 63 6f 6e 20 61 6e 64 20 65 67 67   am bacon and
egg
        0020   73                                               s
    context = {
        '_' : {}
        'bar' : 115
    }
    stack = [
        {
            'data' : 'spam spam spam spam bacon and eggs'
            'self' : Struct('foo')
        }
        {
```

```
            'self' : Struct('foo')
            'stream' : <cStringIO.StringI object at 0x0097A230>
        }
        {
            'context' : {
                '_' : {}
                'bar' : 115
            }
            'obj' : Container:
                bar = 115
            'sc' : Probe('<unnamed 1>')
            'self' : Struct('foo')
            'stream' : <cStringIO.StringI object at 0x0097A230>
            'subobj' : 115
        }
        {
            'context' : {
                '_' : {}
                'bar' : 115
            }
            'self' : Probe('<unnamed 1>')
            'stream' : <cStringIO.StringI object at 0x0097A230>
        }
    ]
Container(bar = 115, baz = 112)
```

4.10.3 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.

```
>>> foo = Struct("foo",
...     UBInt8("bar"),
...     Debugger(
...         Enum(UBInt8("spam"),
...             ABC = 1,
...             DEF = 2,
...             GHI = 3,
...         )
...     ),
...     UBInt8("eggs"),
... )
>>>
>>>
>>> print foo.parse("\x01\x02\x03")
Container:
    bar = 1
    spam = 'DEF'
    eggs = 3
>>>
>>> print foo.parse("\x01\x04\x03")
Debugging exception of MappingAdapter('spam'):
```

```
  File "d:\projects\construct\debug.py", line 112, in _parse
    return self.subcon._parse(stream, context)
  File "d:\projects\construct\core.py", line 174, in _parse
    return self._decode(self.subcon._parse(stream, context), context)
  File "d:\projects\construct\adapters.py", line 77, in _decode
    raise MappingError("no decoding mapping for %r"  % (obj,))
MappingError: no decoding mapping for 4

(you can set the value of 'self.retval', which will be returned)
> d:\projects\construct\adapters.py(77)_decode()
-> raise MappingError("no decoding mapping for %r"  % (obj,))
(Pdb)
(Pdb) u
> d:\projects\construct\core.py(174)_parse()
-> return self._decode(self.subcon._parse(stream, context), context)
(Pdb) u
> d:\projects\construct\debug.py(115)_parse()
-> self.handle_exc("(you can set the value of 'self.retval', "
(Pdb)
(Pdb) l
110         def _parse(self, stream, context):
111             try:
112                 return self.subcon._parse(stream, context)
113             except:
114                 self.retval = NotImplemented
115  ->             self.handle_exc("(you can set the value of 'self.retval',
"
116                     "which will be returned)")
117                 if self.retval is NotImplemented:
118                     raise
119                 else:
120                     return self.retval
(Pdb)
(Pdb) self.retval = "QWERTY"
(Pdb) q
Container:
    bar = 1
    spam = 'QWERTY'
    eggs = 3
>>>
```

CHAPTER 5

API Reference

5.1 construct.core – Core data structures

class construct.core.**Construct**(*name*, *flags=0*)

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.

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()

There is also a flag API:

- _set_flag()
- _clear_flag()
- _inherit_flags()
- _is_flag()

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, and so are names starting with a less-than character ("<"). 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 inherited by default, from inner subconstructs to outer constructs. The enclosing construct may set new flags or clear existing ones, as necessary.

For example, if `FLAG_COPY_CONTEXT` is set, repeaters will pass a copy of the context for each iteration, which is necessary for OnDemand parsing.

parse(*data*)
Parse an in-memory buffer.

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

parse_stream(*stream*)
Parse a stream.

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

build(*obj*)
Build an object in memory.

build_stream(*obj*, *stream*)
Build an object directly into a stream.

sizeof(*context=None*)
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** – contextual data

Returns int of the length of this construct

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

class construct.core.**Subconstruct**(*subcon*)
Abstract subconstruct (wraps an inner construct, inheriting its name and flags).

Subconstructs wrap an inner Construct, inheriting its name and flags.

Parameters **subcon** – the construct to wrap

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

Adapters should implement `_decode()` and `_encode()`.

Parameters **subcon** – the construct to wrap

class construct.core.**StaticField**(*name*, *length*)
A fixed-size byte field.

Parameters

- **name** – field name
- **length** – number of bytes in the field

class construct.core.**FormatField**(*name*, *endianity*, *format*)
A field that uses `struct` to pack and unpack data.

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

Parameters

- **name** – name of the field

- **endianness** – format endianness string; one of "<", ">", or "="
- **format** – a single format character

class construct.core.**MetaField**(*name*, *lengthfunc*)
A variable-length field. The length is obtained at runtime from a function.

Parameters

- **name** – name of the field
- **lengthfunc** – callable that takes a context and returns length as an int

Example:

```
>>> foo = Struct("foo",
...     Byte("length"),
...     MetaField("data", lambda ctx: ctx["length"])
... )
>>> foo.parse("\x03ABC")
Container(data = 'ABC', length = 3)
>>> foo.parse("\x04ABCD")
Container(data = 'ABCD', length = 4)
```

class construct.core.**MetaArray**(*countfunc*, *subcon*)
An array (repeater) of a meta-count. The array will iterate exactly `countfunc()` times. Will raise ArrayError if less elements are found.

See also:

The *Array()* macro, *Range()* and *RepeatUntil()*.

Parameters

- **countfunc** – a function that takes the context as a parameter and returns the number of elements of the array (count)
- **subcon** – the subcon to repeat `countfunc()` times

Example:

```
MetaArray(lambda ctx: 5, UBInt8("foo"))
```

class construct.core.**Range**(*mincount*, *maxcout*, *subcon*)
A range-array. The subcon will iterate between `mincount` to `maxcount` times. If less than `mincount` elements are found, raises RangeError.

See also:

The *GreedyRange()* and *OptionalGreedyRange()* macros.

The general-case repeater. Repeats the given unit for at least `mincount` times, and up to `maxcount` times. If an exception occurs (EOF, validation error), the repeater exits. If less than `mincount` units have been successfully parsed, a RangeError is raised.

Note: This object requires a seekable stream for parsing.

Parameters

- **mincount** – the minimal count
- **maxcount** – the maximal count

- **subcon** – the subcon to repeat

Example:

```
>>> c = Range(3, 7, UBInt8("foo"))
>>> c.parse("\x01\x02")
Traceback (most recent call last):
  ...
construct.core.RangeError: expected 3..7, found 2
>>> c.parse("\x01\x02\x03")
[1, 2, 3]
>>> c.parse("\x01\x02\x03\x04\x05\x06")
[1, 2, 3, 4, 5, 6]
>>> c.parse("\x01\x02\x03\x04\x05\x06\x07")
[1, 2, 3, 4, 5, 6, 7]
>>> c.parse("\x01\x02\x03\x04\x05\x06\x07\x08\x09")
[1, 2, 3, 4, 5, 6, 7]
>>> c.build([1,2])
Traceback (most recent call last):
  ...
construct.core.RangeError: expected 3..7, found 2
>>> c.build([1,2,3,4])
'\x01\x02\x03\x04'
>>> c.build([1,2,3,4,5,6,7,8])
Traceback (most recent call last):
  ...
construct.core.RangeError: expected 3..7, found 8
```

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, context) and returns True if the stop-condition is met, or False to continue.
- **subcon** – the subcon to repeat.

Example:

```
# will read chars until '\x00' (inclusive)
RepeatUntil(lambda obj, ctx: obj == b"\x00",
    Field("chars", 1)
)
```

class construct.core.**Struct**(*name*, **subcons*, ***kw*)

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

See also:

The *Embedded()* macro.

Parameters

- **name** – the name of the structure
- **subcons** – a sequence of subconstructs that make up this structure.

- **nested** – a keyword-only argument that indicates whether this struct creates a nested context. The default is True. This parameter is considered "advanced usage", and may be removed in the future.

Example:

```
Struct("foo",
    UBInt8("first_element"),
    UBInt16("second_element"),
    Padding(2),
    UBInt8("third_element"),
)
```

class construct.core.**Sequence**(*name*, **subcons*, ***kw*)

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

See also:

The *Embedded()* macro.

Parameters

- **name** – the name of the structure
- **subcons** – a sequence of subconstructs that make up this structure.
- **nested** – a keyword-only argument that indicates whether this struct creates a nested context. The default is True. This parameter is considered "advanced usage", and may be removed in the future.

Example:

```
Sequence("foo",
    UBInt8("first_element"),
    UBInt16("second_element"),
    Padding(2),
    UBInt8("third_element"),
)
```

class construct.core.**Union**(*name*, *master*, **subcons*, ***kw*)

a set of overlapping fields (like unions in C). when parsing, all fields read the same data; when building, only the first subcon (called "master") is used.

Parameters

- **name** – the name of the union
- **master** – the master subcon, i.e., the subcon used for building and calculating the total size
- **subcons** – additional subcons

Example:

```
Union("what_are_four_bytes",
    UBInt32("one_dword"),
    Struct("two_words", UBInt16("first"), UBInt16("second")),
    Struct("four_bytes",
        UBInt8("a"),
        UBInt8("b"),
        UBInt8("c"),
        UBInt8("d")
```

```
        ),
    )
```

class construct.core.**Switch**(*name*, *keyfunc*, *cases*, *default=NoDefault('No default value specified')*, *include_key=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.

See also:

Pass().

Parameters

- **name** – the name of the construct
- **keyfunc** – a function that takes the context and returns a key, which will be used to choose the relevant case.
- **cases** – a dictionary mapping keys to constructs. the keys can be any values that may be returned by keyfunc.
- **default** – a default value 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. You can use the builtin construct Pass for 'do-nothing'.
- **include_key** – whether or not to include the key in the return value of parsing. defualt is False.

Example:

```
    Struct("foo",
        UBInt8("type"),
        Switch("value", lambda ctx: ctx.type, {
                1 : UBInt8("spam"),
                2 : UBInt16("spam"),
                3 : UBInt32("spam"),
                4 : UBInt64("spam"),
            }
        ),
    )
```

class construct.core.**Select**(*name*, **subcons*, ***kw*)

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

Note: Requires a seekable stream.

Parameters

- **name** – the name of the construct
- **subcons** – the subcons to try (order-sensitive)
- **include_name** – a keyword only argument, indicating whether to include the name of the selected subcon in the return value of parsing. default is false.

Example:

```
Select("foo",
    UBInt64("large"),
    UBInt32("medium"),
    UBInt16("small"),
    UBInt8("tiny"),
)
```

class construct.core.**Pointer**(*offsetfunc*, *subcon*)
Changes the stream position to a given offset, where the construction should take place, and restores the stream position when finished.

See also:

Anchor(), *OnDemand()* and the *OnDemandPointer()* macro.

Note: Requires a seekable stream.

Parameters

- **offsetfunc** – a function that takes the context and returns an absolute stream position, where the construction would take place
- **subcon** – the subcon to use at `offsetfunc()`

Example:

```
Struct("foo",
    UBInt32("spam_pointer"),
    Pointer(lambda ctx: ctx.spam_pointer,
        Array(5, UBInt8("spam"))
    )
)
```

class construct.core.**Peek**(*subcon*, *perform_build=False*)
Peeks at the stream: parses without changing the stream position. See also Union. If the end of the stream is reached when peeking, returns None.

Note: Requires a seekable stream.

Parameters

- **subcon** – the subcon to peek at
- **perform_build** – whether or not to perform building. by default this parameter is set to False, meaning building is a no-op.

Example:

```
Peek(UBInt8("foo"))
```

class construct.core.**OnDemand**(*subcon*, *advance_stream=True*, *force_build=True*)
Allows for on-demand (lazy) parsing. When parsing, it will return a LazyContainer that represents a pointer to the data, but does not actually parses it from stream until it's "demanded". By accessing the 'value' property of LazyContainers, you will demand the data from the stream. The data will be parsed and cached for later use. You can use the 'has_value' property to know whether the data has already been demanded.

See also:

The *OnDemandPointer()* macro.

Note: Requires a seekable stream.

Parameters

- **subcon** – the subcon to read/write on demand
- **advance_stream** – whether or not to advance the stream position. by default this is True, but if subcon is a pointer, this should be False.
- **force_build** – whether or not to force build. If set to False, and the LazyContainer has not been demaned, building is a no-op.

Example:

```
OnDemand(Array(10000, UBInt8("foo"))
```

class construct.core.**Buffered**(*subcon*, *decoder*, *encoder*, *resizer*)

Creates an in-memory buffered stream, which can undergo encoding and decoding prior to being passed on to the subconstruct.

See also:

The *Bitwise()* macro.

Warning: Do not use pointers inside Buffered.

Parameters

- **subcon** – the subcon which will operate on the buffer
- **encoder** – a function that takes a string and returns an encoded string (used after building)
- **decoder** – a function that takes a string and returns a decoded string (used before parsing)
- **resizer** – a function that takes the size of the subcon and "adjusts" or "resizes" it according to the encoding/decoding process.

Example:

```
Buffered(BitField("foo", 16),
    encoder = decode_bin,
    decoder = encode_bin,
    resizer = lambda size: size / 8,
)
```

class construct.core.**Restream**(*subcon*, *stream_reader*, *stream_writer*, *resizer*)

Wraps the stream with a read-wrapper (for parsing) or a write-wrapper (for building). The stream wrapper can buffer the data internally, reading it from- or writing it to the underlying stream as needed. For example, BitStreamReader reads whole bytes from the underlying stream, but returns them as individual bits.

See also:

The *Bitwise()* macro.

When the parsing or building is done, the stream's close method will be invoked. It can perform any finalization needed for the stream wrapper, but it must not close the underlying stream.

Warning: Do not use pointers inside `Restream`.

Parameters

- **subcon** – the subcon
- **stream_reader** – the read-wrapper
- **stream_writer** – the write wrapper
- **resizer** – a function that takes the size of the subcon and "adjusts" or "resizes" it according to the encoding/decoding process.

Example:

```
Restream(BitField("foo", 16),
    stream_reader = BitStreamReader,
    stream_writer = BitStreamWriter,
    resizer = lambda size: size / 8,
)
```

class `construct.core.`**Reconfig**(*name*, *subcon*, *setflags=0*, *clearflags=0*)
Reconfigures a subconstruct. Reconfig can be used to change the name and set and clear flags of the inner subcon.

Parameters

- **name** – the new name
- **subcon** – the subcon to reconfigure
- **setflags** – the flags to set (default is 0)
- **clearflags** – the flags to clear (default is 0)

Example:

```
Reconfig("foo", UBInt8("bar"))
```

class `construct.core.`**Anchor**(*name*, *flags=0*)
Gets the *anchor* (stream position) at a point in a Construct.

Anchors are useful for adjusting relative offsets to absolute positions, or to measure sizes of Constructs.

To get an absolute pointer, use an Anchor plus a relative offset. To get a size, place two Anchors and measure their difference.

Parameters **name** – the name of the anchor

Note: Anchor Requires a seekable stream, or at least a tellable stream; it is implemented using the `tell()` method of file-like objects.

See also:

Pointer()

class `construct.core.`**Value**(*name*, *func*)
A computed value.

Parameters

- **name** – the name of the value
- **func** – a function that takes the context and return the computed value

Example:

```
Struct("foo",
    UBInt8("width"),
    UBInt8("height"),
    Value("total_pixels", lambda ctx: ctx.width * ctx.height),
)
```

class construct.core.**LazyBound**(*name*, *bindfunc*)

Lazily bound construct, useful for constructs that need to make cyclic references (linked-lists, expression trees, etc.).

Parameters

- **name** – the name of the construct
- **bindfunc** – the function (called without arguments) returning the bound construct

Example:

```
foo = Struct("foo",
    UBInt8("bar"),
    LazyBound("next", lambda: foo),
)
```

construct.core.**Pass = Pass(None)**

A do-nothing construct, useful as the default case for Switch, or to indicate Enums.

See also:

Switch() and the *Enum()* macro.

Note: This construct is a singleton. Do not try to instatiate it, as it will not work.

Example:

```
Pass
```

construct.core.**Terminator = Terminator(None)**

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

Note:

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

•This construct is a singleton. Do not try to instatiate it, as it will not work.

Example:

```
Terminator
```

class construct.core.**ULInt24**(*name*)

A custom made construct for handling 3-byte types as used in ancient file formats. A better implementation would be writing a more flexable version of FormatField, rather then specifically implementing it for this case

5.2 construct.adapters – Adapters

class construct.adapters.**BitIntegerAdapter**(*subcon*, *width*, *swapped=False*, *signed=False*, *bytesize=8*)

Adapter for bit-integers (converts bitstrings to integers, and vice versa). See BitField.

Parameters

- **subcon** – the subcon to adapt
- **width** – the size of the subcon, in bits
- **swapped** – whether to swap byte order (little endian/big endian). default is False (big endian)
- **signed** – whether the value is signed (two's complement). the default is False (unsigned)
- **bytesize** – number of bits per byte, used for byte-swapping (if swapped). default is 8.

class construct.adapters.**MappingAdapter**(*subcon*, *decoding*, *encoding*, *decdefault=NotImplemented*, *encdefault=NotImplemented*)

Adapter that maps objects to other objects. See SymmetricMapping and Enum.

Parameters

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

class construct.adapters.**FlagsAdapter**(*subcon*, *flags*)

Adapter for flag fields. Each flag is extracted from the number, resulting in a FlagsContainer object. Not intended for direct usage. See FlagsEnum.

Parameters

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

class construct.adapters.**StringAdapter**(*subcon*, *encoding=None*)

Adapter for strings. Converts a sequence of characters into a python string, and optionally handles character encoding. See String.

Parameters

- **subcon** – the subcon to convert
- **encoding** – the character encoding name (e.g., "utf8"), or None to return raw bytes (usually 8-bit ASCII).

class construct.adapters.**PaddedStringAdapter**(*subcon*, *padchar='x00'*, *paddir='right'*, *trimdir='right'*)

Adapter for padded strings. See String.

Parameters

- **subcon** – the subcon to adapt
- **padchar** – the padding character. default is "x00".
- **paddir** – the direction where padding is placed ("right", "left", or "center"). the default is "right".
- **trimdir** – the direction where trimming will take place ("right" or "left"). the default is "right". trimming is only meaningful for building, when the given string is too long.

class construct.adapters.**LengthValueAdapter**(*subcon*)
Adapter for length-value pairs. It extracts only the value from the pair, and calculates the length based on the value. See PrefixedArray and PascalString.

Parameters **subcon** – the subcon returning a length-value pair

class construct.adapters.**CStringAdapter**(*subcon*, *terminators='x00'*, *encoding=None*)
Adapter for C-style strings (strings terminated by a terminator char).

Parameters

- **subcon** – the subcon to convert
- **terminators** – a sequence of terminator chars. default is "x00".
- **encoding** – the character encoding to use (e.g., "utf8"), or None to return raw-bytes. the terminator characters are not affected by the encoding.

class construct.adapters.**TunnelAdapter**(*subcon*, *inner_subcon*)
Adapter for tunneling (as in protocol tunneling). A tunnel is construct nested upon another (layering). For parsing, the lower layer first parses the data (note: it must return a string!), then the upper layer is called to parse that data (bottom-up). For building it works in a top-down manner; first the upper layer builds the data, then the lower layer takes it and writes it to the stream.

Parameters

- **subcon** – the lower layer subcon
- **inner_subcon** – the upper layer (tunneled/nested) subcon

Example:

```
# a pascal string containing compressed data (zlib encoding), so first
# the string is read, decompressed, and finally re-parsed as an array
# of UBInt16
TunnelAdapter(
    PascalString("data", encoding = "zlib"),
    GreedyRange(UBInt16("elements"))
)
```

class construct.adapters.**ExprAdapter**(*subcon*, *encoder*, *decoder*)
A generic adapter that accepts '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
- **decoder** – a function that takes (obj, context) and returns an decoded version of obj

Example:

```
    ExprAdapter(UBInt8("foo"),
        encoder = lambda obj, ctx: obj / 4,
        decoder = lambda obj, ctx: obj * 4,
    )
```

class construct.adapters.**HexDumpAdapter**(*subcon*, *linesize=16*)
Adapter for hex-dumping strings. It returns a HexString, which is a string

class construct.adapters.**ConstAdapter**(*subcon*, *value*)
Adapter for enforcing a constant value ("magic numbers"). When decoding, the return value is checked; when building, the value is substituted in.

Parameters

- **subcon** – the subcon to validate
- **value** – the expected value

Example:

```
    Const(Field("signature", 2), "MZ")
```

class construct.adapters.**SlicingAdapter**(*subcon*, *start*, *stop=None*)
Adapter for slicing a list (getting a slice from that list)

Parameters

- **subcon** – the subcon to slice
- **start** – start index
- **stop** – stop index (or None for up-to-end)
- **step** – step (or None for every element)

class construct.adapters.**IndexingAdapter**(*subcon*, *index*)
Adapter for indexing a list (getting a single item from that list)

Parameters

- **subcon** – the subcon to index
- **index** – the index of the list to get

class construct.adapters.**PaddingAdapter**(*subcon*, *pattern='x00'*, *strict=False*)
Adapter for padding.

Parameters

- **subcon** – the subcon to pad
- **pattern** – the padding pattern (character). default is "x00"
- **strict** – whether or not to verify, during parsing, that the given padding matches the padding pattern. default is False (unstrict)

class construct.adapters.**Validator**(*subcon*)
Abstract class: validates a condition on the encoded/decoded object. Override _validate(obj, context) in deriving classes.

Parameters **subcon** – the subcon to validate

class construct.adapters.**OneOf**(*subcon*, *valids*)
Validates that the object is one of the listed values.

Parameters

- **subcon** – object to validate
- **valids** – a set of valid values

Example:

```
>>> OneOf(UBInt8("foo"), [4,5,6,7]).parse("\x05")
5
>>> OneOf(UBInt8("foo"), [4,5,6,7]).parse("\x08")
Traceback (most recent call last):
    ...
construct.core.ValidationError: ('invalid object', 8)
>>>
>>> OneOf(UBInt8("foo"), [4,5,6,7]).build(5)
'\x05'
>>> OneOf(UBInt8("foo"), [4,5,6,7]).build(9)
Traceback (most recent call last):
    ...
construct.core.ValidationError: ('invalid object', 9)
```

class construct.adapters.**NoneOf**(*subcon*, *invalids*)

Validates that the object is none of the listed values.

Parameters

- **subcon** – object to validate
- **invalids** – a set of invalid values

Example:

```
>>> NoneOf(UBInt8("foo"), [4,5,6,7]).parse("\x08")
8
>>> NoneOf(UBInt8("foo"), [4,5,6,7]).parse("\x06")
Traceback (most recent call last):
    ...
construct.core.ValidationError: ('invalid object', 6)
```

5.3 construct.macros – Macros

construct.macros.**Field**(*name*, *length*)

A field consisting of a specified number of bytes.

Parameters

- **name** – the name of the field
- **length** – the length of the field. the length can be either an integer (StaticField), or a function that takes the context as an argument and returns the length (MetaField)

construct.macros.**BitField**(*name*, *length*, *swapped=False*, *signed=False*, *bytesize=8*)

BitFields, as the name suggests, are fields that operate on raw, unaligned bits, and therefore must be enclosed in a BitStruct. Using them is very similar to all normal fields: they take a name and a length (in bits).

Parameters

- **name** – name of the field
- **length** – number of bits in the field, or a function that takes the context as its argument and returns the length

- **swapped** – whether the value is byte-swapped
- **signed** – whether the value is signed
- **bytesize** – number of bits per byte, for byte-swapping

Example:

```
>>> foo = BitStruct("foo",
...     BitField("a", 3),
...     Flag("b"),
...     Padding(3),
...     Nibble("c"),
...     BitField("d", 5),
... )
>>> foo.parse("\xe1\x1f")
Container(a = 7, b = False, c = 8, d = 31)
>>> foo = BitStruct("foo",
...     BitField("a", 3),
...     Flag("b"),
...     Padding(3),
...     Nibble("c"),
...     Struct("bar",
...             Nibble("d"),
...             Bit("e"),
...     )
... )
>>> foo.parse("\xe1\x1f")
Container(a = 7, b = False, bar = Container(d = 15, e = 1), c = 8)
```

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

A padding field (value is discarded)

Parameters

- **length** – the length of the field. the length can be either an integer, or a function that takes the context as an argument and returns the length
- **pattern** – the padding pattern (character) to use. default is "x00"
- **strict** – whether or not to raise an exception is the actual padding pattern mismatches the desired pattern. default is False.

construct.macros.**Flag**(*name*, *truth=1*, *falsehood=0*, *default=False*)

A flag.

Flags are usually used to signify a Boolean value, and this construct maps values onto the `bool` type.

Note: This construct works with both bit and byte contexts.

Warning: Flags default to False, not True. This is different from the C and Python way of thinking about truth, and may be subject to change in the future.

Parameters

- **name** – field name
- **truth** – value of truth (default 1)
- **falsehood** – value of falsehood (default 0)
- **default** – default value (default False)

construct.macros.**Bit**(*name*)
A 1-bit BitField; must be enclosed in a BitStruct

construct.macros.**Nibble**(*name*)
A 4-bit BitField; must be enclosed in a BitStruct

construct.macros.**Octet**(*name*)
An 8-bit BitField; must be enclosed in a BitStruct

construct.macros.**UBInt8**(*name*)
Unsigned, big endian 8-bit integer

construct.macros.**UBInt16**(*name*)
Unsigned, big endian 16-bit integer

construct.macros.**UBInt32**(*name*)
Unsigned, big endian 32-bit integer

construct.macros.**UBInt64**(*name*)
Unsigned, big endian 64-bit integer

construct.macros.**SBInt8**(*name*)
Signed, big endian 8-bit integer

construct.macros.**SBInt16**(*name*)
Signed, big endian 16-bit integer

construct.macros.**SBInt32**(*name*)
Signed, big endian 32-bit integer

construct.macros.**SBInt64**(*name*)
Signed, big endian 64-bit integer

construct.macros.**ULInt8**(*name*)
Unsigned, little endian 8-bit integer

construct.macros.**ULInt16**(*name*)
Unsigned, little endian 16-bit integer

construct.macros.**ULInt32**(*name*)
Unsigned, little endian 32-bit integer

construct.macros.**ULInt64**(*name*)
Unsigned, little endian 64-bit integer

construct.macros.**SLInt8**(*name*)
Signed, little endian 8-bit integer

construct.macros.**SLInt16**(*name*)
Signed, little endian 16-bit integer

construct.macros.**SLInt32**(*name*)
Signed, little endian 32-bit integer

construct.macros.**SLInt64**(*name*)
Signed, little endian 64-bit integer

construct.macros.**UNInt8**(*name*)
Unsigned, native endianity 8-bit integer

construct.macros.**UNInt16**(*name*)
Unsigned, native endianity 16-bit integer

construct.macros.**UNInt32**(*name*)
Unsigned, native endianity 32-bit integer

construct.macros.**UNInt64**(*name*)
Unsigned, native endianity 64-bit integer

construct.macros.**SNInt8**(*name*)
Signed, native endianity 8-bit integer

construct.macros.**SNInt16**(*name*)
Signed, native endianity 16-bit integer

construct.macros.**SNInt32**(*name*)
Signed, native endianity 32-bit integer

construct.macros.**SNInt64**(*name*)
Signed, native endianity 64-bit integer

construct.macros.**BFloat32**(*name*)
Big endian, 32-bit IEEE floating point number

construct.macros.**LFloat32**(*name*)
Little endian, 32-bit IEEE floating point number

construct.macros.**NFloat32**(*name*)
Native endianity, 32-bit IEEE floating point number

construct.macros.**BFloat64**(*name*)
Big endian, 64-bit IEEE floating point number

construct.macros.**LFloat64**(*name*)
Little endian, 64-bit IEEE floating point number

construct.macros.**NFloat64**(*name*)
Native endianity, 64-bit IEEE floating point number

construct.macros.**Array**(*count*, *subcon*)
Repeats the given unit a fixed number of times.

Parameters

- **count** – number of times to repeat
- **subcon** – construct to repeat

Example:

```
>>> c = Array(4, UBInt8("foo"))
>>> c.parse("\x01\x02\x03\x04")
[1, 2, 3, 4]
>>> c.parse("\x01\x02\x03\x04\x05\x06")
[1, 2, 3, 4]
>>> c.build([5,6,7,8])
'\x05\x06\x07\x08'
>>> c.build([5,6,7,8,9])
Traceback (most recent call last):
  ...
construct.core.RangeError: expected 4..4, found 5
```

construct.macros.**PrefixedArray**(*subcon*, *length_field=FormatField('length')*)
An array prefixed by a length field.

Parameters

- **subcon** – the subcon to be repeated
- **length_field** – a construct returning an integer

construct.macros.**GreedyRange**(*subcon*)
Repeats the given unit one or more times.

Parameters **subcon** – construct to repeat

Example:

```
>>> from construct import GreedyRange, UBInt8
>>> c = GreedyRange(UBInt8("foo"))
>>> c.parse("\x01")
[1]
>>> c.parse("\x01\x02\x03")
[1, 2, 3]
>>> c.parse("\x01\x02\x03\x04\x05\x06")
[1, 2, 3, 4, 5, 6]
>>> c.parse("")
Traceback (most recent call last):
  ...
construct.core.RangeError: expected 1..2147483647, found 0
>>> c.build([1,2])
'\x01\x02'
>>> c.build([])
Traceback (most recent call last):
  ...
construct.core.RangeError: expected 1..2147483647, found 0
```

construct.macros.**OptionalGreedyRange**(*subcon*)
Repeats the given unit zero or more times. This repeater can't fail, as it accepts lists of any length.

Parameters **subcon** – construct to repeat

Example:

```
>>> from construct import OptionalGreedyRange, UBInt8
>>> c = OptionalGreedyRange(UBInt8("foo"))
>>> c.parse("")
[]
>>> c.parse("\x01\x02")
[1, 2]
>>> c.build([])
''
>>> c.build([1,2])
'\x01\x02'
```

construct.macros.**Optional**(*subcon*)
An optional construct. if parsing fails, returns None.

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

construct.macros.**Bitwise**(*subcon*)
Converts the stream to bits, and passes the bitstream to subcon

Parameters **subcon** – a bitwise construct (usually BitField)

construct.macros.**Aligned**(*subcon*, *modulus=4*, *pattern='\x00'*)
Aligns subcon to modulus boundary using padding pattern

Parameters

- **subcon** – the subcon to align
- **modulus** – the modulus boundary (default is 4)
- **pattern** – the padding pattern (default is x00)

construct.macros.**SeqOfOne**(*name*, **args*, ***kw*)
A sequence of one element. only the first element is meaningful, the rest are discarded

Parameters

- **name** – the name of the sequence
- ***args** – subconstructs
- ****kw** – any keyword arguments to Sequence

construct.macros.**Embedded**(*subcon*)
Embeds a struct into the enclosing struct.

Parameters **subcon** – the struct to embed

construct.macros.**Rename**(*newname*, *subcon*)
Renames an existing construct

Parameters

- **newname** – the new name
- **subcon** – the subcon to rename

construct.macros.**Alias**(*newname*, *oldname*)
Creates an alias for an existing element in a struct

Parameters

- **newname** – the new name
- **oldname** – the name of an existing element

construct.macros.**SymmetricMapping**(*subcon*, *mapping*, *default=NotImplemented*)
Defines a symmetrical mapping: a->b, b->a.

Parameters

- **subcon** – the subcon to map
- **mapping** – the encoding mapping (a dict); the decoding mapping is achieved by reversing this mapping
- **default** – the default value to use when no mapping is found. if no default value is given, and exception is raised. setting to Pass would return the value "as is" (unmapped)

construct.macros.**Enum**(*subcon*, ***kw*)
A set of named values mapping.

Parameters

- **subcon** – the subcon to map
- ****kw** – 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* to pass the unmapped value as-is

construct.macros.**FlagsEnum**(*subcon*, ***kw*)
A set of flag values mapping.

Parameters

- **subcon** – the subcon to map
- ****kw** – keyword arguments which serve as the encoding mapping

construct.macros.**AlignedStruct**(*name*, **subcons*, ***kw*)
A struct of aligned fields

Parameters

- **name** – the name of the struct
- ***subcons** – the subcons that make up this structure
- ****kw** – keyword arguments to pass to Aligned: 'modulus' and 'pattern'

construct.macros.**BitStruct**(*name*, **subcons*)
A struct of bitwise fields

Parameters

- **name** – the name of the struct
- ***subcons** – the subcons that make up this structure

construct.macros.**EmbeddedBitStruct**(**subcons*)
An embedded BitStruct. no name is necessary.

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

construct.macros.**String**(*name*, *length*, *encoding=None*, *padchar=None*, *paddir='right'*, *trimdir='right'*)
A configurable, fixed-length string field.

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

Parameters

- **name** – name
- **length** – length, in bytes
- **encoding** – encoding (e.g. "utf8") or None for no encoding
- **padchar** – optional character to pad out strings
- **paddir** – direction to pad out strings; one of "right", "left", or "both"
- **trim** (*str*) – direction to trim strings; one of "right", "left"

Example:

```
>>> from construct import String
>>> String("foo", 5).parse("hello")
'hello'
>>>
>>> String("foo", 12, encoding = "utf8").parse("hello joh\xd4\x83n")
u'hello joh\u0503n'
>>>
>>> foo = String("foo", 10, padchar = "X", paddir = "right")
>>> foo.parse("helloXXXXX")
'hello'
>>> foo.build("hello")
'helloXXXXX'
```

construct.macros.**PascalString**(*name*, *length_field=FormatField('length')*, *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 appear in the same Container as the PascalString, with the given name.

Parameters

- **name** – name
- **length_field** – a field which will store the length of the string
- **encoding** – encoding (e.g. "utf8") or None for no encoding

Example:

```
>>> foo = PascalString("foo")
>>> foo.parse("\x05hello")
'hello'
>>> foo.build("hello world")
'\x0bhello world'
>>>
>>> foo = PascalString("foo", length_field = UBInt16("length"))
>>> foo.parse("\x00\x05hello")
'hello'
>>> foo.build("hello")
'\x00\x05hello'
```

construct.macros.**CString**(*name*, *terminators='\x00'*, *encoding=None*, *char_field=StaticField(None)*)

A string ending in a terminator.

CString is similar to the strings of C, C++, and other related programming languages.

By default, the terminator is the NULL byte (b''0x00'').

Parameters

- **name** – name
- **terminators** – sequence of valid terminators, in order of preference
- **encoding** – encoding (e.g. "utf8") or None for no encoding
- **char_field** – construct representing a single character

Example:

```
>>> foo = CString("foo")
>>> foo.parse(b"hello\x00")
b'hello'
>>> foo.build(b"hello")
b'hello\x00'
>>> foo = CString("foo", terminators = b"XYZ")
>>> foo.parse(b"helloX")
b'hello'
>>> foo.parse(b"helloY")
b'hello'
>>> foo.parse(b"helloZ")
b'hello'
>>> foo.build(b"hello")
b'helloX'
```

construct.macros.**GreedyString**(*name*, *encoding=None*, *char_field=StaticField(None)*)
A configurable, variable-length string field.

Parameters

- **name** – name
- **encoding** – encoding (e.g. "utf8") or None for no encoding
- **char_field** – construct representing a single character

Example:

```
>>> foo = GreedyString("foo")
>>> foo.parse(b"hello\x00")
b'hello\x00'
>>> foo.build(b"hello\x00")
b'hello\x00'
>>> foo.parse(b"hello")
b'hello'
>>> foo.build(b"hello")
b'hello'
```

construct.macros.**IfThenElse**(*name*, *predicate*, *then_subcon*, *else_subcon*)
An if-then-else conditional construct: if the predicate indicates True, *then_subcon* will be used; otherwise *else_subcon*

Parameters

- **name** – the name of the construct
- **predicate** – a function taking the context as an argument and returning True or False
- **then_subcon** – the subcon that will be used if the predicate returns True
- **else_subcon** – the subcon that will be used if the predicate returns False

construct.macros.**If**(*predicate*, *subcon*, *elsevalue=None*)
An if-then conditional construct: if the predicate indicates True, subcon will be used; otherwise, *elsevalue* will be returned instead.

Parameters

- **predicate** – a function taking the context as an argument and returning True or False
- **subcon** – the subcon that will be used if the predicate returns True
- **elsevalue** – the value that will be used should the predicate return False. by default this value is None.

construct.macros.**OnDemandPointer**(*offsetfunc*, *subcon*, *force_build=True*)
An on-demand pointer.

Parameters

- **offsetfunc** – a function taking the context as an argument and returning the absolute stream position
- **subcon** – the subcon that will be parsed from the *offsetfunc()* stream position on demand
- **force_build** – see OnDemand. by default True.

construct.macros.**Magic**(*data*)
A 'magic number' construct. It is used for file signatures, to validate that the given pattern exists. When parsed. the value must match.

Parameters **data** – a bytes object

Example:

```
elf_header = Struct("elf_header",
    Magic("ELF"),
    # ...
)
```

5.4 construct.debug – Debugging

Debugging utilities for constructs

class construct.debug.**Probe**(*name=None*, *show_stream=True*, *show_context=True*, *show_stack=True*, *stream_lookahead=100*)

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

See also:

Debugger.

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("foo",
    UBInt8("a"),
    Probe("between a and b"),
    UBInt8("b"),
)
```

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 on-the-fly).

Parameters **subcon** – the subcon to debug

Example:

```
Debugger(
    Enum(UBInt8("foo"),
        a = 1,
        b = 2,
        c = 3
    )
)
```

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

C

construct.adapters, 47
construct.core, 37
construct.debug, 59
construct.macros, 50

A

Adapter (class in construct.core), 38
Alias() (in module construct.macros), 55
Aligned() (in module construct.macros), 54
AlignedStruct() (in module construct.macros), 56
Anchor (class in construct.core), 45
Array() (in module construct), 15
Array() (in module construct.macros), 53

B

BFloat32() (in module construct.macros), 53
BFloat64() (in module construct.macros), 53
Bit() (in module construct.macros), 52
BitField() (in module construct), 17
BitField() (in module construct.macros), 50
BitIntegerAdapter (class in construct.adapters), 47
BitStruct() (in module construct.macros), 56
Bitwise() (in module construct.macros), 54
Buffered (class in construct.core), 44
build() (construct.core.Construct method), 38
build_stream() (construct.core.Construct method), 38

C

ConstAdapter (class in construct.adapters), 49
Construct (class in construct.core), 37
construct.adapters (module), 47
construct.core (module), 37
construct.debug (module), 59
construct.macros (module), 50
CString() (in module construct), 27
CString() (in module construct.macros), 57
CStringAdapter (class in construct.adapters), 48

D

Debugger (class in construct.debug), 59

E

Embedded() (in module construct.macros), 55
EmbeddedBitStruct() (in module construct.macros), 56
Enum() (in module construct.macros), 55
ExprAdapter (class in construct.adapters), 48

F

Field() (in module construct), 23
Field() (in module construct.macros), 50
Flag() (in module construct.macros), 51
FlagsAdapter (class in construct.adapters), 47
FlagsEnum() (in module construct.macros), 55
FormatField (class in construct.core), 38

G

GreedyRange() (in module construct), 15
GreedyRange() (in module construct.macros), 54
GreedyString() (in module construct.macros), 57

H

HexDumpAdapter (class in construct.adapters), 49

I

If() (in module construct.macros), 58
IfThenElse() (in module construct.macros), 58
IndexingAdapter (class in construct.adapters), 49

L

LazyBound (class in construct.core), 46
LengthValueAdapter (class in construct.adapters), 48
LFloat32() (in module construct.macros), 53
LFloat64() (in module construct.macros), 53

M

Magic() (in module construct.macros), 58
MappingAdapter (class in construct.adapters), 47
MetaArray (class in construct.core), 39
MetaField (class in construct.core), 39

N

NFloat32() (in module construct.macros), 53
NFloat64() (in module construct.macros), 53
Nibble() (in module construct.macros), 52
NoneOf (class in construct), 21

NoneOf (class in construct.adapters), 50

O

Octet() (in module construct.macros), 52
OnDemand (class in construct.core), 43
OnDemandPointer() (in module construct.macros), 58
OneOf (class in construct), 21
OneOf (class in construct.adapters), 49
Optional() (in module construct.macros), 54
OptionalGreedyRange() (in module construct), 15
OptionalGreedyRange() (in module construct.macros), 54

P

PaddedStringAdapter (class in construct.adapters), 47
Padding() (in module construct.macros), 51
PaddingAdapter (class in construct.adapters), 49
parse() (construct.core.Construct method), 38
parse_stream() (construct.core.Construct method), 38
PascalString() (in module construct), 27
PascalString() (in module construct.macros), 56
Pass (in module construct.core), 46
Peek (class in construct.core), 43
Pointer (class in construct.core), 43
PrefixedArray() (in module construct.macros), 53
Probe (class in construct.debug), 59

R

Range (class in construct.core), 39
Range() (in module construct), 14
Reconfig (class in construct.core), 45
Rename() (in module construct.macros), 55
RepeatUntil (class in construct.core), 40
Restream (class in construct.core), 44

S

SBInt16() (in module construct.macros), 52
SBInt32() (in module construct.macros), 52
SBInt64() (in module construct.macros), 52
SBInt8() (in module construct.macros), 52
Select (class in construct.core), 42
SeqOfOne() (in module construct.macros), 55
Sequence (class in construct.core), 41
sizeof() (construct.core.Construct method), 38
SlicingAdapter (class in construct.adapters), 49
SLInt16() (in module construct.macros), 52
SLInt32() (in module construct.macros), 52
SLInt64() (in module construct.macros), 52
SLInt8() (in module construct.macros), 52
SNInt16() (in module construct.macros), 53
SNInt32() (in module construct.macros), 53
SNInt64() (in module construct.macros), 53
SNInt8() (in module construct.macros), 53
StaticField (class in construct.core), 38
String() (in module construct), 26
String() (in module construct.macros), 56
StringAdapter (class in construct.adapters), 47
Struct (class in construct.core), 40
Subconstruct (class in construct.core), 38
Switch (class in construct.core), 42
SymmetricMapping() (in module construct.macros), 55

T

Terminator (in module construct.core), 46
TunnelAdapter (class in construct.adapters), 48

U

UBInt16() (in module construct.macros), 52
UBInt32() (in module construct.macros), 52
UBInt64() (in module construct.macros), 52
UBInt8() (in module construct.macros), 52
ULInt16() (in module construct.macros), 52
ULInt24 (class in construct.core), 46
ULInt32() (in module construct.macros), 52
ULInt64() (in module construct.macros), 52
ULInt8() (in module construct.macros), 52
UNInt16() (in module construct.macros), 52
UNInt32() (in module construct.macros), 52
UNInt64() (in module construct.macros), 53
UNInt8() (in module construct.macros), 52
Union (class in construct.core), 41

V

Validator (class in construct.adapters), 49
Value (class in construct.core), 45