The Basics

Fields

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

Some examples of parsing:

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

Some examples of building:

```
>>> from construct import Int16ub, Int16sb
>>> Int16ub.build(31337)
b'zi'
>>> Int16sb.build(-31337)
b'\x85\x97'
```

Other fields like:

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

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

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

Variable-length fields

```
>>> VarInt.build(1234567890)
b'\xd2\x85\xd8\xcc\x04'
>>> VarInt.sizeof()
SizeofError: Error in path (sizeof)
```

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

```
>>> Int16ub[2].sizeof()
4
>>> VarInt[2].sizeof()
SizeofError: Error in path (sizeof)
```

Structs

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

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

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

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

There is another declaration syntax that uses keyword arguments. Truth be told, I am not keen on using this way of declaring Structs. You should use the `/` operator as shown in first example.

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

Operator `+` can also be used to make Structs. Structs are nested when added. Truth be told, I am not keen on using this way of declaring Structs either.

```
>>> d = "a"/Byte + "inner"/Struct("b"/Byte) + "c"/Byte
```

Containers

What is that `Container` object, anyway? Well, a `Container` is a subclass of `dict`. They provide pretty-printing and allow to access items as attributes as well as keys, and they also preserve insertion order. `ListContainer`, similarly, is a subclass of `list`. Both `Container` and `ListContainer` provide searching functionality. Let's see more of those:

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

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

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

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

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

Note that not all parameters can be accessed via attribute access (dot operator). If the name of an item matches a method name of the `Container`, it can only be accessed via key access (square brackets operator). This includes the following names: `clear`, `copy`, `fromkeys`, `get`, `items`, `keys`, `move_to_end`, `pop`, `popitem`, `search`, `search_all`, `setdefault`, `update`, `values`.

```
>>> x = Container(update=5)
>>> x["update"]
5
>>> x.update  # not usable via dot access
<bound method Container.update of Container(update=5)>
```

Nesting and embedding

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

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

It used to be that Structs could have been embedded (flattened out). However, this created more problems than it solved so this feature was eventually removed. Since Construct 2.10 it is no longer possible to embed structs. You should, and always should have been, be nesting them just like in the example above.

Showing path information in exceptions

If your construct throws an exception, for any reason, there should be a "path information" attached to it. In the example below, the `(parsing) -> a -> b -> c -> foo` field throws an exception due to lack of bytes to consume. You can see that in the exception message.

```
>>> x = Struct(
...     'foo' / Bytes(1),
...     'a' / Struct(
...         'foo' / Bytes(1),
...         'b' / Struct(
...             'foo' / Bytes(1),
...             'c' / Struct(
...                 'foo' / Bytes(1),
...                 'bar' / Bytes(1)
...             )
...         )
...     )
... )
>>> x.parse(b'\xff' * 3)
construct.core.StreamError: Error in path (parsing) -> a -> b -> c -> foo
stream read less than specified amount, expected 1, found 0
```

Note that compiled parsing classes may not provide a path information.

Hidden context entries

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

```
>>> d = Struct(
...     'x' / Computed(1),
...     'inner' / Struct(
...         'inner2' / Struct(
```

```
...             'x' / Computed(this._root.x),
...             'z' / Computed(this._params.z),
...             'zz' / Computed(this._root._.z),
...         ),
...     ),
...     Probe(),
... )
>>> setGlobalPrintPrivateEntries(True)
>>> d.parse(b'', z=2)
--------------------------------------------------
Probe, path is (parsing), into is None
Container:
    _ = Container:
        z = 2
        _parsing = True
        _building = False
        _sizing = False
        _params = <recursion detected>
    _params = Container:
        z = 2
        _parsing = True
        _building = False
        _sizing = False
        _params = <recursion detected>
    _root = <recursion detected>
    _parsing = True
    _building = False
    _sizing = False
    _subcons = Container:
        x = <Renamed x +nonbuild <Computed +nonbuild>>
        inner = <Renamed inner +nonbuild <Struct +nonbuild>>
    _io = <_io.BytesIO object at 0x7fd91e7313b8>
    _index = None
    x = 1
    inner = Container:
        _io = <_io.BytesIO object at 0x7fd91e7313b8>
        inner2 = Container:
            _io = <_io.BytesIO object at 0x7fd91e7313b8>
            x = 1
            z = 2
            zz = 2
--------------------------------------------------
Container(x=1, inner=Container(inner2=Container(x=1, z=2, zz=2)))
```

Explanation is as follows:

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

Sequences

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

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

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

```
>>> d = Int16ub >> Sequence(Byte, Byte)
>>> d.parse(bytes(4))
ListContainer([0, 0, 0])
# it is NOT nested like ListContainer(0, ListContainer(0, 0))
```

Repeaters

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

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

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

`GreedyRange` attempts to parse until EOF or subcon fails to parse correctly. Either way, when `GreedyRange` encounters either failure it seeks the stream back to a position after last successful subcon parsing. This means the stream must be seekable/tellable (doesn't work inside `Bitwise`).

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

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

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

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

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

Processing on-the-fly

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

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

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

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

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