namedstruct Documentation

Release 1.1.1

Hu Bo

January 05, 2017

Contents

Contents:

CHAPTER 1

Quick Guide

Following tutorials describes the basic functions of this library. See *Reference* for more details.

1.1 Basic Tutorial

The common usage of namedstruct is to create **type definitions** with interfaces like *namedstruct.nstruct* and *namedstruct.enum*, use *namedstruct.typedef.parse()* or *namedstruct.typedef.create()* interfaces on the created types to parse a packed struct bytes to a Python object, use *namedstruct.typedef.new()* (or directly *namedstruct.typedef.__call__()* of the created type, like normal Python classes) to create new objects of the defined type, and use `namedstruct.NamedStruct._tobytes()` to convert objects to packed bytes. Use *namedstruct.dump()* to convert the Python object to json-serializable format.

1.1.1 Basic Usage

Create structs with pre-defined primitive types or other defined types:

```
from namedstruct import *
mytype = nstruct((uint16, 'myshort'),  # unsigned short int
                 (uint8, 'mybyte'),       # unsigned char
                 (uint8,),                # a padding byte of unsigned char
                 (char[5], 'mystr'),   # a 16-byte bytes string
                 (uint8,),
                 (uint16[5], 'myarray'),    #
                 name = 'mytype',
                 padding = 1)
# Create an object
obj0 = mytype()
# Access a field
s = obj0.myshort
obj0.myshort = 12
# Create an object with the specified fields initialized
obj1 = mytype(myshort = 2, mystr = b'123', myarray = [1,2,3,4,5])
# Unpack an object from stream, return the object and bytes size used
obj2,size = mytype.parse(b'\x00\x02\x01\x00123\x00\x00\x00\x00\x01\x00\x02\x00\x03\x00\x04\x00\x05')
# Unpack an object from packed bytes
obj3 = mytype.create(b'\x00\x02\x01\x00123\x00\x00\x00\x00\x01\x00\x02\x00\x03\x00\x04\x00\x05')
# Estimate the struct size
size = len(obj0)
# Estimate the struct size excludes automatic padding bytes
```

```
size2 = obj0._realsize()
# Pack the object
b = obj0._tobytes()
b2 = mytype.tobytes(obj0)

# Use the type in other structs

mytype2 = nstruct((mytype, 'mydata'),
                  (mytype[4], 'mystructarray'),
                  name = 'mytype2',
                  padding = 1)

obj4 = mytype2()
obj4.mydata.myshort = 12
obj4.mystructarray[0].mybyte = 7
b3 = obj4._tobytes()
```

The struct is equivalent to C struct definition:

```
typedef struct mytype{
    unsigned short int myshort;
    unsigned char mybyte;
    char _padding;
    unsigned char mystr[5];
    char _padding2;
    unsigned short myarray[5];
}mytype;
```

1.1.2 Variable Length Data Types

Some data types can have variable length. they can consume extra data:

```
myvartype = nstruct((uint16, 'a'),
                    (uint16[0], 'extra'),
                    padding = 1,
                    name = 'myvartype')

"""
>>> myvartype.create(b'\x00\x02').extra
[]
>>> myvartype.create(b'\x00\x02\x00\x01\x00\x003\x00\x05').extra
[1,3,5]
"""

obj1 = myvartype()
obj1.extra.append(5)
obj1.extra.extend((1,2,3))
"""
>>> obj1._tobytes()
b'\x00\x00\x05\x00\x01\x00\x02\x00\x03'
"""

myvartype2 = nstruct((uint16, 'a'),
                    (raw, 'data'),
                    padding = 1,
                    name = 'myvartype2')
```

```
"""
>>> myvartype.create(b'\x00\x01abcde').data
b'abcde'
>>> myvartype(a = 1, data = b'XYZ')._tobytes()
b'\x00\x01XYZ'
"""
```

Variable length data types do not have determined length, so they cannot be parsed correctly themselves. The outer-struct should store some extra informations to determine the correct length. Specify a *size* option to calculate the correct struct size. Most times the total size of the struct is stored in a field, return that value to let the struct parses correctly:

```
from namedstruct import nstruct, uint16, raw, packrealsize
my_unsize_struct = nstruct((uint16, 'length'),
                           (raw, 'data'),
                           padding = 1,
                           name = 'my_unsize_struct')

"""
>>> my_unsize_struct.create(b'\x00\x07abcde').data
b'abcde'
>>> my_unsize_struct.parse(b'\x00\x07abcde')[0].data
b''
"""

my_size_struct = nstruct((uint16, 'length'),
                         (raw, 'data'),
                         padding = 1,
                         name = 'my_size_struct',
                         prepack = packrealsize('length'),
                         size = lambda x: x.length)
"""
packrealsize('length') is equivalent to:

    def _packsize(x):
        x.length = x._realsize()
"""

"""
>>> my_size_struct(data = b'abcde')._tobytes()
b'\x00\x07abcde'
>>> my_size_struct.parse(b'\x00\x07abcde')[0].data
b'abcde'
"""
```

1.1.3 Common Options

name defines the struct name. It is used in many ways, so it is recommended to define a name option for any struct:

```
struct1 = nstruct((uint8,'data'),name = 'struct1')
"""
>>> repr(struct1)
'struct1'
"""
```

init defines the struct initializer. It is a function executed every time a new instance of this type is created with *new* (*packvalue* is a helper function which returns a function setting specified value to corresponding attribute when

executing):

```
from namedstruct import nstruct, uint8, packvalue
struct2 = nstruct((uint8,'data'),name = 'struct2',init = packvalue(1, 'data'))

"""
>>> struct2().data
1
"""
```

prepack is a function executed just before struct pack. It is useful to automatically generate fields related to other fields, like checksum or struct size (*packexpr* is a helper function which evaluates the function with the struct as the first argument and stores the return value to the specified attribute):

```
from namedstruct import nstruct, uint8, packexpr
struct3 = nstruct((uint8, 'a'),
                  (uint8, 'b'),
                  (uint8, 'sum'),
                  name = 'struct3',
                  prepack = packexpr(lambda x: x.a + x.b, 'sum')
                  )
"""
Equivalent to:

    def _prepack_func(x):
        x.sum = x.a + x.b
    struct3 = nstruct((uint8, 'a'),
                      (uint8, 'b'),
                      (uint8, 'sum'),
                      name = 'struct3',
                      prepack = _prepack_func
                      )
"""
```

The fields are in big-endian (network order) by default. To parse or build little-endian struct, specify *endian* option to the struct and use little-endian types:

```
from namedstruct import nstruct, uint16, uint16_le

struct4 = nstruct((uint16, 'a'),
                  (uint16, 'b'),
                  name = 'struct4',
                  padding = 1)

struct4_le = nstruct((uint16_le, 'a'),
                  (uint16_le, 'b'),
                  name = 'struct4_le',
                  padding = 1,
                  endian = '<')

"""
>>> struct4(a = 1, b = 2)._tobytes()
b'\x00\x01\x00\x02'
>>> struct4_le(a = 1, b = 2)._tobytes()
b'\x01\x00\x02\x00'
"""
```

The struct is automatically padded to multiplies of *padding* bytes. By default *padding* = 8, means that the struct is padded to align to 8-bytes (64-bits) boundaries. Set *padding* = 1 to disable padding:

```
struct5 = nstruct((uint16, 'a'),
                  (uint8, 'b'),
                  name = 'struct5')
"""
>>> struct5(a=1,b=2)._tobytes()
b'\x00\x01\x02\x00\x00\x00\x00\x00'
>>> len(struct5(a=1,b=2))
8
>>> struct5(a=1,b=2)._realsize()
3
"""

struct5_p2 = nstruct((uint16, 'a'),
                  (uint8, 'b'),
                  name = 'struct5_p2',
                  padding = 2)

"""
>>> struct5_p4(a=1,b=2)._tobytes()
b'\x00\x01\x02\x00'
>>> len(struct5_p4(a=1,b=2))
4
>>> struct5_p4(a=1,b=2)._realsize()
3
"""

struct5_p1 = nstruct((uint16, 'a'),
                  (uint8, 'b'),
                  name = 'struct5_p1',
                  padding = 1)

"""
>>> struct5_p1(a=1,b=2)._tobytes()
b'\x00\x01\x02'
>>> len(struct5_p1(a=1,b=2))
3
>>> struct5_p1(a=1,b=2)._realsize()
3
"""
```

See *namedstruct.nstruct* for all valid options.

1.1.4 Extend(inherit) defined structs

With *size* option, a struct can have more data than the defined fields on parsing. Also it is possible to let a struct use more data with *namedstruct.typedef.create()*. Besides using the data for variable length fields, it is also possible to use the extra data for extending. The extending works like C/C++ inherits: the fields of base class is parsed first, then the extending fields. Different from C/C++ inherits, the child classe types are automatically determined with *criterias* on parsing:

```
from namedstruct import *
my_base = nstruct((uint16, 'length'),
                  (uint8, 'type'),
                  (uint8, 'basedata'),
                  name = 'my_base',
                  size = lambda x: x.length,
                  prepack = packrealsize('length'),
```

```
                    padding = 4)

my_child1 = nstruct((uint16, 'data1'),
                    (uint8, 'data2'),
                    name = 'my_child1',
                    base = my_base,
                    criteria = lambda x: x.type == 1,
                    init = packvalue(1, 'type'))

my_child2 = nstruct((uint32, 'data3'),
                    name = 'my_child2',
                    base = my_base,
                    criteria = lambda x: x.type == 2,
                    init = packvalue(2, 'type'))

"""
Fields and most base class options are inherited, e.g. size, prepack, padding
>>> my_child1(basedata = 1, data1 = 2, data2 = 3)._tobytes()
b'\x00\x07\x01\x01\x00\x02\x03\x00'
>>> my_child2(basedata = 1, data3 = 4)._tobytes()
b'\x00\x08\x02\x01\x00\x00\x00\x04'
"""

# Fields in child classes are automatically parsed when the type is determined
obj1, _ = my_base.parse(b'\x00\x07\x01\x01\x00\x02\x03\x00')
"""
>>> obj1.basedata
1
>>> obj1.data1
2
>>> obj1.data2
3
>>> obj1._gettype()
my_child1
"""

# Base type can be used in fields or arrays of other structs

my_base_array = nstruct((uint16, 'total_len'),
                        (my_base[0], 'array'),
                        name = 'my_base_array',
                        padding = 1,
                        size = lambda x: x.total_len,
                        prepack = packrealsize('total_len'))

obj2 = my_base_array()
obj2.array.append(my_child1(data1 = 1, data2 = 2, basedata = 3))
obj2.array.append(my_child2(data3 = 4, basedata = 5))
"""
>>> obj2._tobytes()
b'\x00\x12\x00\x07\x01\x03\x00\x01\x02\x00\x00\x08\x02\x05\x00\x00\x00\x04'
"""
obj3, _ = my_base_array.parse(b'\x00\x12\x00\x07\x01\x03\x00\x01\x02\x00\x00\x08\x02\x05\x00\x00\x00
"""
>>> obj3.array[0].data2
2
"""
```

1.1.5 Embedded structs

An anonymous field has different means when it has different types:

- An anonymous field of primitive type (e.g. uint8, uint16, or uint8[5]) is automatically converted to padding bytes
- An anonymous field of struct type is an embedded struct.

When a struct is embedded into another struct, all the fields of the embedded struct act same as directly defined in the parent struct. The embedded struct has the same function as normal structs, like **inherit**, **variable length data**, **padding**, *size* option, and struct size. Parent fields may also be accessed from the embedded struct:

```
inner_struct = nstruct((uint16[0], 'array'),
                       name = 'inner_struct',
                       padding = 4,
                       size = lambda x: x.arraylength,         # Get size from parent struct
                       prepack = packrealsize('arraylength')   # Pack to parent struct
                       )

parent_struct = nstruct((uint16, 'totallength'),
                        (uint16, 'arraylength'),
                        (inner_struct,),
                        (raw, 'extra'),
                        padding = 1,
                        name = 'parent_struct',
                        size = lambda x: x.arraylength,
                        prepack = lambda: packrealsize('totallength'))

"""
>>> parent_struct(array = [1,2,3], extra = b'abc')._tobytes()
b'\x00\x0f\x00\x06\x00\x01\x00\x02\x00\x03\x00\x00abc'
>>> parent_struct.parse(b'\x00\x0f\x00\x06\x00\x01\x00\x02\x00\x03\x00\x00abc')[0].array
[1,2,3]
"""
```

1.1.6 Other Data Types

namedstruct.enum is a way to define C/C++ enumerates. They act like normal primitive types, but when use `nstruct.dump()` to generate human readable result, the values are converted to readable names (or readable name list for *bitwise* enumerates). See *Formatting* for more details on human readable formatting. See *namedstruct.enum* document for usage details.:

```
my_enum = enum('my_enum', globals(), uint16,
               MY_A = 1,
               MY_B = 2,
               MY_C = 3)

my_type = nstruct((my_enum, 'type'),
                  name = 'my_type',
                  padding = 1)

"""
>>> dump(my_type(type=MY_A))
{'_type': '<my_type>', 'type': 'MY_A'}
"""
```

namedstruct.optional is a simpler way to define an optional field. It is actually an small embedded struct, with a criteria to determine whether it should parse the field:

```
struct1 = nstruct((uint8, 'hasdata'),
                  (optional(uint16, 'data', lambda x: x.hasdata),),
                  name = 'struct1',
                  prepack = packexpr(lambda x: hasattr(x, 'data'), 'hasdata'),
                  padding = 1)
"""
>>> struct1()._tobytes()
b'\x00'
>>> struct1(data=2)._tobytes()
b'\x01\x00\x02'
>>> struct1.parse(b'\x01\x00\x02')[0].data
2
"""
```

namedstruct.darray is an array type whose element count is determined by other field. It is another way to store an array with varaible element count:

```
from namedstruct import nstruct, uint16, raw, packrealsize, darray
my_size_struct = nstruct((uint16, 'length'),
                         (raw, 'data'),
                         padding = 1,
                         name = 'my_size_struct',
                         prepack = packrealsize('length'),
                         size = lambda x: x.length)

my_darray_struct = nstruct((uint16, 'arraysize'),
                           (darray(my_size_struct, 'array', lambda x: x.arraysize),),
                           name = 'my_darray_struct',
                           prepack = packexpr(lambda x: len(x.array), 'arraysize'),
                           padding = 1)

"""
>>> my_darray_struct(array = [my_size_struct(data = b'ab'), my_size_struct(data = b'cde')])._tobytes
b'\x00\x02\x00\x04ab\x00\x05cde'
>>> my_darray_struct.parse(b'\x00\x02\x00\x04ab\x00\x05cde')[0].array[1].data
b'cde'
"""
```

namedstruct.bitfield is a mini-struct with bits:

```
mybit = bitfield(uint64,
                 (4, 'first'),
                 (5, 'second'),
                 (2,),    # Padding bits
                 (19, 'third'),    # Can cross byte border
                 (1, 'array', 20), # A array of 20 1-bit numbers
                 name = 'mybit',
                 init = packvalue(2, 'second'))
"""
>>> mybit().second
2
>>> mybit(first = 5, third = 7)._tobytes()
b'Q\x00\x00\x1c\x00\x00\x00\x00'          # b'Q' = b'\x51'
# the uint64 is '0b0101000100000000000000000000011100000000000000000000000000000000'
"""
```

1.2 Formatting

One of the important functions of *namedstruct* library is the ability to convert a binary struct into human readable format. The library does not convert the struct directly into strings. Instead, it converts the struct into dictionaries and lists, which is suitable for *pprint* or *json.dumps* as well as further processing. Call *namedstruct.dump()* on a parsed value to convert it into this format:

```
from namedstruct import *

my_struct = nstruct((uint16, 'x'),
                    (uint8, 'y'),
                    (raw, 'data'),
                    padding = 1,
                    name = 'my_struct')
s = my_struct(x = 1, y = 2, data = b'abc')
"""
>>> dump(s)
{'y': 2, 'x': 1, '_type': '<my_struct>', 'data': 'abc'}
>>> dump(s, typeinfo = DUMPTYPE_NONE)
{'y': 2, 'x': 1, 'data': 'abc'}
"""
```

1.2.1 Human Readable Formatting

There are different options on calling *namedstruct.dump()*. When *humanread* is True, extra formatting procedures are executed on special data types to convert them into human-readable format. For example, enumerates defined by *namedstruct.enum* are converted into names:

```
from namedstruct import *

gender = enum('gender', globals(),uint8,
              MALE = 0,
              FEMALE = 1)

abilities = enum('abilities', globals(), uint16, True,
                 SWIMMING = 1<<0,
                 JUMPING = 1<<1,
                 RUNNING = 1<<2,
                 CLIMBING = 1<<3)

person = nstruct((cstr, 'name'),
                 (gender, 'gender'),
                 (abilities, 'abilities'),
                 name = 'person',
                 padding = 1)

john = person(name = 'john', gender = MALE, abilities = JUMPING | CLIMBING)

"""
>>> dump(john, True)
{'gender': 'MALE', '_type': '<person>', 'abilities': 'JUMPING CLIMBING', 'name': 'john'}
>>> dump(john, False)
{'gender': 0, '_type': '<person>', 'abilities': 10, 'name': 'john'}
"""
```

See *namedstruct.dump()* for descriptions of parameters.

1.2.2 Customized Human Readable Formatting

There are three approaches to customize the display format of a data type when calling *namedstruct.dump()*.

- Formatter Attribute

  Set the *formatter* attribute of a defined data type to a function to execute it on formatting. The function should take the "unformatted" representation of the data as a parameter, and return the formatted data:

```
ETH_ALEN = 6
mac_addr = uint8[ETH_ALEN]
mac_addr.formatter = lambda x: ':'.join('%02X' % (n,) for n in x)

my_packet = nstruct((mac_addr, 'src'),
                    (mac_addr, 'dst'),
                    name = 'my_packet',
                    padding = 1)


p = my_packet(src = [0x00, 0xFF, 0x31, 0x14, 0x25, 0x17], dst = [0x00, 0xFF, 0x12, 0x45, 0x7a, 0
"""
>>> dump(p, False)
{'src': [0, 255, 49, 20, 37, 23], 'dst': [0, 255, 18, 69, 122, 11], '_type': '<my_packet>'}
>>> dump(p)
{'src': '00:FF:31:14:25:17', 'dst': '00:FF:12:45:7A:0B', '_type': '<my_packet>'}
"""
```

  Notice that the *formatter* attribute of the data type only has effect when the data type is used as the type of a field in a struct.

- Formatter Option

  namedstruct.nstruct has a *formatter* option, which is similar to the *formatter* attribute. Different from setting the attribute, the option is executed even if the data type is the out-most struct:

```
ops = ['+', '-', '*', '/']

expr = nstruct((uint32, 'a'),
               (uint32, 'b'),
               (uint8, 'op'),
               name = 'expr',
               padding = 1,
               formatter = lambda x: '%d %s %d' % (x['a'], ops[x['op']], x['b']))

"""
>>> dump(expr(a = 12, b = 23, op = 0))
'12 + 23'
"""
```

- Type Extending

  Sometimes it is not possible to know the exact format of a field in a struct until the struct is subclassed. *extend* option can be used to replace the formatting procedure of a field as if the field is in type defined by *extend* option:

```
from namedstruct import *

person_type = enum('person_type', globals(), uint8,
                   STUDENT = 0,
                   TEACHER = 1)

student_flag = enum('student_flag', globals(), uint8, True,
```

```
                        HARDWORKING = 1<<0,
                        SMART = 1<<1,
                        FRIENDLY = 1<<2,
                        STRONG = 1<<3)

    teacher_flag = enum('teacher_flag', globals(), uint8, True,
                        HARDWORKING = 1<<0,
                        EXPERIENCED = 1<<1,
                        ENTHUSIASTIC = 1<<2,
                        STRICT = 1<<3)

    person = nstruct((uint16, 'length'),
                     (person_type, 'type'),
                     (uint8, 'flag'),
                     classifier = lambda x: x.type,
                     name = 'person',
                     padding = 1,
                     size = lambda x: x.length,
                     prepack = packrealsize('length'))

    student = nstruct((uint8, 'grade'),
                      (uint8, 'classno'),
                      name = 'student',
                      classifyby = (STUDENT,),
                      init = packvalue(STUDENT, 'type'),
                      extend = {'flag': student_flag},
                      base = person)

    teacher = nstruct((uint8, 'age'),
                      (uint8, 'subject'),
                      name = 'teacher',
                      classifyby = (TEACHER,),
                      init = packvalue(TEACHER, 'type'),
                      extend = {'flag': teacher_flag},
                      base = person)

    """
    >>> dump(student(grade = 2, classno = 1, flag = 10))
    {'_type': '<student>', 'grade': 2, 'length': 0, 'flag': 'SMART STRONG', 'classno': 1, 'type': 'S
    >>> dump(teacher(age = 35, subject = 0, flag = 10))
    {'_type': '<teacher>', 'age': 35, 'flag': 'EXPERIENCED STRICT', 'length': 0, 'type': 'TEACHER',
    """
```

The overall formatting procedure of a struct is in this order:

1. Dump result of every field (including fields of base type, fields of embedded structs) is calculated. If the field value is a struct, the struct formatting is the same as this procedure.
2. Fields defined in this struct (including fields of base type, excluding fields of embedded structs) is formatted with the "formatter", either from the original type or from the *extend* type. If any descendant fields are extended with *extend*, they are also formatted.
3. Embedded structs are formatted like in 2 i.e. fields with "formatter" and fields in *extend* are formatted.
4. *formatter* option is executed if it is defined in the struct type.

1.3 Advanced Examples

The key to create a very complicated dynamic data type is using embedded struct and extended struct wisely. An embedded struct can access fields of parent struct and keep its own characteristics at the same time; An extended struct can access fields of the base struct and also inherits the characteristics of the base type.

1.3.1 Struct with More Than One Variable Length Fields

Use embedded struct to control the variable length fields (a simplied version of ARP packet with some fields removed):

```
from namedstruct import *

_arp_hw_src = nstruct((raw, 'hw_src'),
                      size = lambda x: x.hw_length,
                      prepack = packrealsize('hw_length'),
                      name = '_arp_hw_src')

_arp_hw_dst = nstruct((raw, 'hw_dst'),
                      size = lambda x: x.hw_length,
                      name = '_arp_hw_dst')

_arp_nw_src = nstruct((raw, 'nw_src'),
                      size = lambda x: x.nw_length,
                      prepack = packrealsize('nw_length'),
                      name = '_arp_nw_src')

_arp_nw_dst = nstruct((raw, 'nw_dst'),
                      size = lambda x: x.nw_length,
                      name = '_arp_nw_dst')

arp = nstruct((uint16, 'hw_length'),
              (uint16, 'nw_length'),
              (_arp_hw_src,),
              (_arp_nw_src,),
              (_arp_hw_dst,),
              (_arp_nw_dst,),
              name = 'arp',
              padding = 1
              )

"""
>>> arp(hw_src = b'\x00\xff\x01\x3f\x11\x1b', hw_dst = b'\x00\xff\x08\x7e\x10\x0a', nw_src = b'\xc0\x
        nw_dst = b'\xc0\xa8\x01\x03')._tobytes()
b'\x00\x06\x00\x04\x00\xff\x01?\x11\x1b\xc0\xa8\x01\x02\x00\xff\x08~\x10\n\xc0\xa8\x01\x03'
>>> dump(arp.parse(b'\x00\x03\x00\x01\x00\xff\xaa\x00\xff\xbb\x17\x19')[0])
{'_type': '<arp>', 'nw_length': 1, 'nw_dst': '\x19', 'hw_src': '\x00\xff\xaa', 'nw_src': '\x00', 'hw_
"""
```

The length of *hw_src*, *hw_dst*, *nw_src*, *nw_dst* fields are determined by *hw_length* and *nw_length*. It is also possible to define the embedded struct inside the main struct:

```
arp = nstruct((uint16, 'hw_length'),
              (uint16, 'nw_length'),
              (nstruct((raw, 'hw_src'),
                   size = lambda x: x.hw_length,
                   name = '_arp_hw_src',
                   prepack = packrealsize('hw_length'),
```

```
                padding = 1),),
            (nstruct((raw, 'nw_src'),
                size = lambda x: x.nw_length,
                name = '_arp_nw_src',
                prepack = packrealsize('nw_length'),
                padding = 1),),
            (nstruct((raw, 'hw_dst'),
                size = lambda x: x.hw_length,
                name = '_arp_hw_dst',
                padding = 1),),
            (nstruct((raw, 'nw_dst'),
                size = lambda x: x.nw_length,
                name = '_arp_nw_dst',
                padding = 1),),
            name = 'arp',
            padding = 1
            )
```

1.3.2 Extend an Embedded Struct

An embedded struct can also be extended (inherited) (a simplied version of ARP packet with l2 header and some fields removed):

```
from namedstruct import *

ETH_ALEN = 6
mac_addr = uint8[ETH_ALEN]
mac_addr.formatter = lambda x: ':'.join('%02X' % (n,) for n in x)

ether_l2 = nstruct((mac_addr, 'dmac'),
                   (mac_addr, 'smac'),
                   (uint16, 'ethertype'),
                   name = 'ether_l2',
                   padding = 1,
                   size = lambda x: 18 if x.ethertype == 0x8100 else 14)

ether_l2_8021q = nstruct((bitfield(uint16,
                                   (3, 'pri'),
                                   (1, 'cfi'),
                                   (12, 'tag')),),
                         (uint16, 'ethertype2'),
                         base = ether_l2,
                         name = 'ether_l2_8021q',
                         criteria = lambda x: x.ethertype == 0x8100,
                         init = packvalue(0x8100, 'ethertype'))

ether_l3 = nstruct((ether_l2,),
                   name = 'ether_l3',
                   padding = 1,
                   classifier = lambda x: getattr(x, 'ethertype2', x.ethertype))

arp = nstruct((uint16, 'hw_length'),
            (uint16, 'nw_length'),
            (nstruct((raw, 'hw_src'),
                size = lambda x: x.hw_length,
                name = '_arp_hw_src',
                prepack = packrealsize('hw_length'),
```

```
                padding = 1),),
            (nstruct((raw, 'nw_src'),
                size = lambda x: x.nw_length,
                name = '_arp_nw_src',
                prepack = packrealsize('nw_length'),
                padding = 1),),
            (nstruct((raw, 'hw_dst'),
                size = lambda x: x.hw_length,
                name = '_arp_hw_dst',
                padding = 1),),
            (nstruct((raw, 'nw_dst'),
                size = lambda x: x.nw_length,
                name = '_arp_nw_dst',
                padding = 1),),
            name = 'arp',
            padding = 1
            )

ether_l3_arp = nstruct((arp,),
                       name = 'ether_l3_arp',
                       base = ether_l3,
                       classifyby = (0x0806,),
                       init = packvalue(0x0806, 'ethertype'))

"""
# Create a packet without VLAN tag
>>> ether_l3_arp(dmac = [0x00, 0xff, 0x1a, 0x1b, 0x1c, 0x1d],
                 smac = [0x00, 0xff, 0x0a, 0x0b, 0x0c, 0x0d],
                 hw_src = b'\x00\xff\x0a\x0b\x0c\x0d',
                 hw_dst = b'\x00\xff\x1a\x1b\x1c\x1d',
                 nw_src = b'\xc0\xa8\x01\x02',
                 nw_dst = b'\xc0\xa8\x01\x03')._tobytes()
b'\x00\xff\x1a\x1b\x1c\x1d\x00\xff\n\x0b\x0c\r\x08\x06\x00\x06\x00\x04\x00\xff\n\x0b\x0c\r\xc0\xa8\x

# Create a packet with VLAN tag
>>> ether_l3_arp((ether_l2, ether_l2_8021q),
                 dmac = [0x00, 0xff, 0x1a, 0x1b, 0x1c, 0x1d],
                 smac = [0x00, 0xff, 0x0a, 0x0b, 0x0c, 0x0d],
                 hw_src = b'\x00\xff\x0a\x0b\x0c\x0d',
                 hw_dst = b'\x00\xff\x1a\x1b\x1c\x1d',
                 nw_src = b'\xc0\xa8\x01\x02',
                 nw_dst = b'\xc0\xa8\x01\x03',
                 tag = 100,
                 ethertype2 = 0x0806)._tobytes()
b'\x00\xff\x1a\x1b\x1c\x1d\x00\xff\n\x0b\x0c\r\x81\x00\x00d\x08\x06\x00\x06\x00\x04\x00\xff\n\x0b\x0

# Parse a packet without VLAN tag
>>> dump(ether_l3.create(b'\x00\xff\x1a\x1b\x1c\x1d\x00\xff\n\x0b\x0c\r\x08\x06\x00\x06\x00\x04\x00\
{'dmac': '00:FF:1A:1B:1C:1D', '_type': '<ether_l3_arp>', 'nw_length': 4, 'nw_dst': '\xc0\xa8\x01\x03

# Parse a packet with VLAN tag
>>> dump(ether_l3.create(b'\x00\xff\x1a\x1b\x1c\x1d\x00\xff\n\x0b\x0c\r\x81\x00\x00d\x08\x06\x00\x06
{'dmac': '00:FF:1A:1B:1C:1D', '_type': '<ether_l3_arp>', 'nw_length': 4, 'nw_dst': '\xc0\xa8\x01\x03
"""
```

CHAPTER 2

Reference

2.1 Public API Interfaces

class namedstruct.**typedef**

Base class for type definitions. Types defined with *nstruct*, *prim*, *optional*, *bitfield* all have these interfaces.

__call__(**args*, ***kwargs*)

Same as new()

__getitem__(*size*)

Same as array(size), make it similar to C/Java type array definitions. It is worth to notice that the array is in column order, unlike in C. For example, uint16[3][4] is a list with 4 elements, each is a list of 3 uint16. uint16[5][0] is a variable length array, whose elements are lists of 5 uint16.

__weakref__

list of weak references to the object (if defined)

array(*size*)

Create an array type, with elements of this type. Also available as indexer([]), so mytype[12] creates a array with fixed size 12; mytype[0] creates a variable size array. :param size: the size of the array, 0 for variable size array.

create(*buffer*)

Create a object from all the bytes. If there are additional bytes, they may be fed greedily to a variable length type, or may be used as "extra" data. :param buffer: bytes of a packed struct. :returns: an object with exactly the same bytes when packed. :raises: BadFormatError or BadLenError if the bytes cannot completely form this type.

formatdump(*dumpvalue*, *v*)

Format the dumpvalue when dump() is called with humanread = True

Parameters

- **dumpvalue** – the return value of dump() before formatting.
- **v** – the original data

Returns return a formatted version of the value.

inline()

Returns whether this type can be "inlined" into other types. If the type is inlined into other types, it is splitted and re-arranged to form a FormatParser to improve performance. If not, the parser is used instead. :returns: None if this type cannot be inlined; a format definition similar to FormatParser if it can.

isextra()
Returns whether this type can take extra data. For example, a variable size array will be empty when parsed, but will have elements when fed with extra data (like when create() is called)

new(**args*, ***kwargs*)
Create a new object of this type. It is also available as __call__, so you can create a new object just like creating a class instance: a = mytype(a=1,b=2)

Parameters

- **args** – Replace the embedded struct type. Each argument is a tuple (name, newtype). It is equivalent to call _replace_embedded_type with *name* and *newtype* one by one. Both the "directly" embedded struct and the embedded struct inside another embedded struct can be set. If you want to replace an embedded struct in a replaced struct type, make sure the outer struct is replaced first. The embeded struct type must have a *name* to be replaced by specify *name* option.
- **kwargs** – extra key-value arguments, each one will be set on the new object, to set value to the fields conveniently.

Returns a new object, with the specified "kwargs" set.

parse(*buffer*)
Parse the type from specified bytes stream, and return the first one if exists.

Parameters **buffer** – bytes from a stream, may contains only part of the struct, exactly one struct, or additional bytes after the struct.

Returns None if the data is incomplete; (data, size) else, where data is the parsed data, size is the used bytes length, so the next struct begins from buffer[size:]

parser()
Get parser for this type. Create the parser on first call.

tobytes(*obj*)
Convert the object to packed bytes. If the object is a NamedStruct, it is usually obj._tobytes(); but it is not possible to call _tobytes() for primitive types.

vararray()
Same as array(0)

class namedstruct.**nstruct**(**members*, ***arguments*)
Generic purpose struct definition. Struct is defined by fields and options, for example:

```
mystruct = nstruct((uint32, 'a'),
                   (uint16[2], 'b'),
                   (mystruct2,'c'),
                   (uint8,),
                   (mystruct3,),
                   name = 'mystruct')
```

uint32, uint16, uint8 are standard types from *stdprim*. uint16[2] creates an array type with fixed size of 2. Field names must be valid attribute names, and CANNOT begin with '_', because names begin with '_' is preserved for internal use. The defined struct is similar to C struct:

```
typedef struct{
    int a;
    short b[2];
    struct mystruct2 c;
    char _padding;
    struct mystruct3;
} mystruct;
```

A struct is in big-endian (network order) by default. If you need a little-endian struct, specify *endian* option to '<' and use little-endian version of types at the same time:

```
mystruct_le = nstruct((uint32_le, 'a'),
                      (uint16_le[2], 'b'),
                      (mystruct2_le,'c'),
                      (uint8_le,),
                      (mystruct3_le,),
                      name = 'mystruct_le',
                      endian = '<')
```

Fields can be named or anonymous. Following rules are applied: - A named field parses a specified type and set the result to attribute with the specified name

- A anonymous struct field embeds the struct into this struct: every attribute of the embedded struct can be accessed from the parent struct, and every attribute of parent struct can also be accessed from the embedded struct.
- Any anonymous primitive types act as padding bytes, so there is not a specialized padding type
- Anonymous array is not allowed.

Structs are aligned to 8-bytes (*padding* = 8) boundaries by default, so if a struct defines fields of only 5 bytes, 3 extra padding bytes are appended to the struct automatically. The struct is always padded to make the size multiplies of "padding" even if it contains complex sub-structs or arrays, so it is more convenient than adding padding bytes to definitions. For example, if a struct contains a variable array of 7-bytes size data type, you must add 1 byte when the variable array has only one element, and add 2 bytes when the variable array has two elements. The alignment (padding) can be adjusted with *padding* option. Specify 4 for 4-bytes (32-bit) alignment, 2 for 2-bytes (16-bit), etc. Specify 1 for 1-bytes alignment, which equals to disable alignment.

Structs can hold more bytes than the size of defined fields. A struct created with create() takes all the input bytes as part of the struct. The extra bytes are used in variable length data types, or stored as "extra" data to serve automatic sub-class.

A variable length data type is a kind of data type whose size cannot be determined by itself. For example, a variable array can have 0 - infinite elements, and the actual size cannot be determined by the serialized bytes. If you merge multiple variable arrays into one bytes stream, the boundary of each array disappears. These kinds of data types usually can use as much data as possible. A struct contains a variable length data type as the last field type is also variable length. For example:

```
varray = nstruct((uint16, 'length'),
                 (mystruct[0], 'structs'),
                 name = 'varray')
```

When use parse() to parse from a bytes stream, the length of "structs" array is always 0, because parse() takes bytes only when they are ensured to be part of the struct. When use create() to create a struct from bytes, all the following bytes are parsed into "structs" array, which creates a array with correct length.

Usually we want to be able to unpack the packed bytes and get exactly what are packed, so it is necessary to have a way to determine the struct size, for example, from the struct size stored in a field before the variable size part. Set *size* option to a function to retrieve the actual struct size:

```
varray2 = nstruct((uint16, 'length'),
                  (mystruct[0], 'structs'),
                  name = 'varray2',
                  size = lambda x: x.length,
                  prepack = packrealsize('length')
                  )
```

When this struct is parsed from a bytes stream, it is parsed in three steps: - A first parse to get the non-variable parts of the struct, just like what is done when *size* is not set

- *size* function is called on the parsed result, and returns the struct size
- A second parse is done with the correct size of data, like what is done with create()

Usually the *size* function should return "real" size, which means size without the padding bytes at the end of struct, but it is usually OK to return the padded size. Variable length array ignores extra bytes which is not enough to form a array element. If the padding bytes are long enough to form a array element, the parsed result may contain extra empty elements and must be processed by user.

The *prepack* option is a function which is executed just before pack, so the actual size is automatically stored to 'length' field. There is no need to care about the 'length' field manually. Other processes may also be done with *prepack* option.

A struct without variable length fields can also have the *size* option to preserve data for extension. The bytes preserved are called "extra" data and stored in "_extra" attribute of the parsed result. You may use the "extra" data with _getextra() and _setextra() interface of the returned NamedStruct object. On packing, the "extra" data is appended directly after the last field, before the "padding" bytes, and counts for struct size, means the "extra" data is preserved in packing and unpacking:

```
base1 = nstruct((uint16, 'length'),
                (uint8, 'type'),
                (uint8,),
                name = 'base1',
                size = lambda x: x.length,
                prepack = packrealsize('length'))
```

The extra data can be used in sub-class. A sub-classed struct is a struct begins with the base struct, and use the "extra" data of the base struct as the data of the extended fields. It works like the C++ class derive:

```
child1 = nstruct((uint16, 'a1'),
                 (uint16, 'b1'),
                 base = base1,
                 criteria = lambda x: x.type == 1,
                 init = packvalue(1, 'type'),
                 name = 'child1')
child2 = nstruct((uint8, 'a2'),
                 (uint32, 'b2'),
                 base = base1,
                 criteria = lambda x: x.type == 2,
                 init = packvalue(2, 'type'),
                 name = 'child2')
```

A "child1" struct consists of a uint16 'length', a uint8 'type', a uint8 padding byte, a uint16 'a1', a uint16 'b1'; the "child2" struct consists of a uint16 'length', a uint8 'type', a uint8 padding byte, a uint8 'a2', a uint32 'b2', and 7 padding bytes (to make the total size 16-bytes).

criteria option determines when will the base class be sub-classed into this type. It is a function which takes the base class parse result as the paramter, and return True if the base class should be sub-classed into this type. *init* is the struct initializer. When create a new struct with new(), all fields will be initialized to 0 if *init* is not specified; if *init* is specified, the function is executed to initialize some fields to a pre-defined value. Usually sub-classed types need to initialize a field in the base class to identity the sub-class type, so it is common to use *init* together with *criteria*.

The packed bytes of both "child1" and "child2" can be parsed with type "base1". The parsing is done with the following steps: - base1 is parsed, *size* is executed and extra data is stored in "_extra" attribute

- every *criteria* of sub-class types is executed until one returns True
- extended fields of the sub-class struct is created with "extra" data of base1, and the _extra attribute is removed, just like create() on a struct without base class.

•if none of the *criteria* returns True, the base1 struct is unchanged.

If there are a lot of sub-class types, executes a lot of *criteria* is a O(n) procedure and is not very efficient. The base class may set *classifier* option, which is executed and returns a hashable value. The sub-class type set *classifyby* option, which is a tuple of valid values. If the return value of *classifier* matches any value in *classifyby*, the base class is sub-classed to that type. The procedure is O(1).

The size of a sub-classed type is determined by the base type. If the base type has no "extra" bytes, it cannot be sub-classed. If the base type does not have a *size* option, it is not possible to parse the struct and sub-class it from a bytes stream. But it is still possible to use create() to create the struct and automatically sub-class it. It is useless to set *size* option on a struct type with base type, though not harmful. *prepack* can still be used, and will be executed in base first, then the sub-class type.

If there are still bytes not used by the sub-class type, they are stored as the "extra" data of the sub-class type, so the sub-class type can be sub-classed again. Also _getextra and _setextra can be used.

Every time a base class is parsed, the same steps are done, so it is possible to use a base class as a field type, or even a array element type:

```
base1array = nstruct((uint16, 'length'),
                     (base1[0], 'items'),
                     size = lambda x: x.length,
                     prepack = packrealsize('length'),
                     name = 'base1array')
```

The base1array type can take any number of child1 and child2 in "items" with any order, even if they have different size.

It is often ambiguous for a struct with additional bytes: they can be "extra" data of the struct itself, or they can be "extra" data of last field. Set *lastextra* option to strictly specify how they are treated. When *lastextra* is True, the "extra" data is considered to be "extra" data of last field if possible; when *lastextra* is False, the "extra" data is considered to be "extra" data of the struct itself. If nether is specified (or set None), the *lastextra* is determined by following rules:

- A type is a variable length type if *isextra* of the type returns True. In particular, variable length arrays, raw bytes(*raw*, *varchr*) are variable length types.
- **A struct type is a variable length type if all the following criterias met:**
  - it does not have a *base* type
  - it does not have a *size* option
  - *lastextra* is True (either strictly specified or automatically determined)
- If the last field can be "inlined" (see option *inline*), *lastextra* is False (even if you strictly specify True)
- The struct *lastextra* is True if the last field of the struct is a variable length type

In summary, a struct *lastextra* is True if the last field is variable length, and itself is also variable length if no *base* or *size* is specified. For complicated struct definitions, you should consider strictly specify it. A struct with *lastextra* = True cannot have sub-class types since it does not have "extra" bytes.

A simple enough struct may be "inlined" into another struct to improve performance, which means it is splitted into fields, and merged to another struct which has a field of this type. Option *inline* controls whether the struct should be "inlined" into other structs. By default, *inline* is automatically determined by the number of fields. Types with *base*, *size*, *prepack*, or *init* set will not be inlined. You should strictly specify *inline* = False when: - A inner struct has different endian with outer struct - The struct is a base type but *size* is not defined

formatter and *extend* options have effect on human readable dump result. When you call dump() on a NamedStruct, or a value with a NamedStruct inside, and specify humanread = True, the dump result is converted to a

human readable format. The *formatter* option is a function which takes the dump result of this struct (a dictionary) as a parameter. The fields in the dump result are already formatted by the type definitions. the *formatter* should modify the dump result and return the modified result. The *formatter* option is not inherited to sub-classed types, so if the struct is sub-classed, the *formatter* is not executed. The *formatter* option will not be executed in a embedded struct, because the struct shares data with the parent struct and cannot be formatted itself.

extend is another method to modify the output format. If a type defines a "formatter" method, it is used by the parent struct when a struct with fields of this type is formatted. The "formatter" method only affects fields in structs because a primitive value (like a integer) does not have enough information on the original type. You may add customized formatters to a custom type:

```
uint32_hex = prim('I', 'uint32_hex')
uint32_hex.formatter = lambda x: ('0x%08x' % (x,))    # Convert the value to a hex string
```

When a field of this type is defined in a struct, the dump result will be formatted with the "formatter". Elements of an array of this type is also formatted by the "formatter". Notice that an array type can also have a "formatter" defined, in that case the elements are first formatted by the inner type "formatter", then a list of the formatted results are passed to the array "formatter" to format the array as a whole. *enum* types and some pre-defined types like *ip_addr*, *mac_addr* have defined their own "formatter"s.

extend overrides the "formatter" of specified fields. *extend* option is a dictionary, whose keys are field names, or tuple of names (a property path). The values are data types, instances of *nstruct*, *enum* or any other subclasses of *typedef*. If the new data type has a "formatter", the original formatter of that type is replaced by the new formatter i.e. the original formatter is NOT executed and the new formatter will execute with the original dump result. It is often used to override the type of a field in the base type. For example, a base type defineds a "flags" field but the format differs in every sub-classed type, then every sub-classed type may override the "flags" field to a different *enum* to show different result. If the extend type does not have a formatter defined, the original formatter is NOT replaced. Add a formatter which does not modify the result i.e. (lambda x: x) to explicitly disable the original formatter.

It is also convenient to use *extend* with array fields. The "formatter" of array element are overridden to the new element type of the extended array type. If a "formatter" is defined on the array, it also overrides the original array "formatter".

You may also override fields of a named field or anonymous field. Specify a key with a tuple like (f1, f2, f3...) will override the field self.f1.f2.f3. But notice that the original formatter is only replaced if the struct is "inlined" into this struct. If it is not, the formatter cannot be replaced since it is not part of the "formatting procedure" of this struct. For named fields, the original formatter is executed before the extended formatter; for anonymous fields, the original formatter is executed after the extended formatter. It is not assured that this order is kept unchanged in future versions. If the original type does not have a formatter, it is safe to extend it at any level.

The *extend* options are inherited by the sub-classed types, so a sub-classed struct will still use the *extend* option defined in the base class. If the sub-classed type overrides a field again with *extend* option, the corresponding override from the base class is replaced again with the formatter from the sub-classed type.

The overall formatting procedure of a struct is in this order:

1. Dump result of every field (including fields of base type, fields of embedded structs) is calculated. If the field value is a struct, the struct formatting is the same as this procedure.
2. Fields defined in this struct (including fields of base type, excluding fields of embedded structs) is formatted with the "formatter", either from the original type or from the *extend* type. If any descendant fields are extended with *extend*, they are also formatted.
3. Embedded structs are formatted like in 2 i.e. fields with "formatter" and fields in *extend* are formatted.
4. *formatter* is executed if it is defined in the struct type.

Structs may also define "formatter" like other types, besides the *formatter* option. It is useful in embedded structs because the *formatter* of embedded struct is not executed. But it is only executed when it is contained in another struct either named or embedded, like in other types.

Exceptions in formatters are always ignored. A debug level information is logged with logging module, you may enable debug logging to view them.

__init__(**members*, ***arguments*)
nstruct initializer, create a new nstruct type

Parameters

- **members** – field definitions, either named or anonymous; named field is a tuple with 2 members (type, name); anonymous field is a tuple with only 1 member (type,)
- **arguments** – optional keyword arguments, see nstruct docstring for more details:

  size A function to retrieve the struct size

  prepack A function to be executed just before packing, usually used to automatically store the struct size to a specified field (with packsize() or packrealsize())

  base This type is a sub-class type of a base type, should be used with criteria and/or classifyby

  criteria A function determines whether this struct (of base type) should be sub-classed into this type

  endian Default to '>' as big endian or "network order". Specify '<' to use little endian.

  padding Default to 8. The struct is automatically padded to align to "padding" bytes boundary, i.e. padding the size to be ((_realsize + (padding - 1)) // padding). Specify 1 to disable alignment.

  lastextra Strictly specify whether the unused bytes should be considered to be the "extra" data of the last field, or the "extra" data of the struct itself. See nstruct docstring for more details.

  name Specify a readable struct name. It is always recommended to specify a name. A warning is generated if you do not specify one.

  inline Specify if the struct can be "inlined" into another struct. See nstruct docstring for details.

  init initializer of the struct, executed when a new struct is created with new()

  classifier defined in a base class to get a classify value, see nstruct docstring for details

  classifyby a tuple of hashable values. The values is inserted into a dictionary to quickly find the correct sub-class type with classify value. See nstruct docstring for details.

  formatter A customized function to modify the human readable dump result. The input parameter is the current dump result; the return value is the modified result, and will replace the current dump result.

  extend Another method to modify the human readable dump result of the struct. It uses the corresponding type to format the specified field, instead of the default type, e.g. extend a uint16 into an enumerate type to show the enumerate name; extend a 6-bytes string to mac_addr_bytes to format the raw data to MAC address format, etc.

class namedstruct.**enum**(*readablename=None*, *namespace=None*, *basefmt='I'*, *bitwise=False*, ***kwargs*)
Enumerate types are extensions to standard primitives. They are exactly same with the base type, only with the

exception that when converted into human readable format with dump(), they are converted to corresponding enumerate names for better readability:

```
myenum = enum('myenum', globals(), uint16,
                MYVALUE1 = 1,
                MYVALUE2 = 2,
                MYVALUE3 = 3)
```

To access the defined enumerate values:

```
v = myenum.MYVALUE1             # 1
```

When globals() is specified as the second parameter, the enumerate names are exported to current module, so it is also accessible from module globals():

```
v = MYVALUE2    # 2
```

If you do not want to export the names, specify None for *namespace* parameter.

A enumerate type can be bitwise, or non-bitwise. Non-bitwise enumerate types stand for values exactly matches the enumerate values. When formatted, they are converted to the corresponding enumerate name, or keep unchanged if there is not a corresponding name with that value. Bitwise enumerate types stand for values that are bitwise OR (|) of zero, one or more enumerate values. When a bitwise enumerate value is formatted, it is formatted like the following example:

```
mybitwise = enum('mybitwise', globals(), uint16, True,
                A = 0x1,
                B = 0x2,
                C = 0x4,
                D = 0x8,
# It is not necessary (though common) to have only 2-powered values. Merge of two or more
# values may stand for a special meaning.
                E = 0x9)

# Formatting:
# 0x1 -> 'A'
# 0x3 -> 'A B'
# 0x8 -> 'D'
# 0x9 -> 'E'    (prefer to match more bits as a whole)
# 0xb -> 'B E'
# 0x1f -> 'B C E 0x10' (extra bits are appended to the sequence in hex format)
# 0x10 -> '0x10'
# 0x0 -> 0      (0 is unchanged)
```

__contains__(*item*)
Test whether a value is defined.

__init__(*readablename=None*, *namespace=None*, *basefmt='I'*, *bitwise=False*, ***kwargs*)
Initializer :param readablename: name of this enumerate type

Parameters

- **namespace** – A dictionary, usually specify globals(). The *kwargs* are updated to this dictionary, so the enumerate names are exported to current globals and you do not need to define them in the module again. None to disable this feature.
- **basefmt** – base type of this enumerate type, can be format strings or a *prim* type
- **bitwise** – if True, the enumerate type is bitwise, and will be formatted to space-separated names; if False, the enumerate type is non-bitwise and will be formatted to a single name.

- **kwargs** – ENUMERATE_NAME = ENUMERATE_VALUE format definitions of enumerate values.

astype(*primtype*, *bitwise=False*)
Create a new enumerate type with same enumerate values but a different primitive type e.g. convert a 16-bit enumerate type to fit in a 32-bit field. :param primtype: new primitive type :param bitwise: whether or not the new enumerate type should be bitwise

extend(*namespace=None*, *name=None*, ***kwargs*)
Create a new enumerate with current values merged with new enumerate values :param namespace: same as __init__ :param name: same as __init__ :param kwargs: same as __init__ :returns: a new enumerate type

formatter(*value*)
Format a enumerate value to enumerate names if possible. Used to generate human readable dump result.

getDict()
Returns a dictionary whose keys are enumerate names, and values are corresponding enumerate values.

getName(*value*, *defaultName=None*)
Get the enumerate name of a specified value. :param value: the enumerate value :param defaultName: returns if the enumerate value is not defined :returns: the corresponding enumerate value or *defaultName* if not found

getValue(*name*, *defaultValue=None*)
Get the enumerate value of a specified name. :param name: the enumerate name :param defaultValue: returns if the enumerate name is not defined :returns: the corresponding enumerate value or *defaultValue* if not found

importAll(*gs*)
Import all the enumerate values from this enumerate to *gs* :param gs: usually globals(), a dictionary. At lease __setitem__ should be implemented if not a dictionary.

merge(*otherenum*)
Return a new enumerate type, which has the same primitive type as this type, and has enumerate values defined both from this type and from *otherenum* :param otherenum: another enumerate type :returns: a new enumerate type

tostr(*value*)
Convert the value to string representation. The formatter is first used, and if the return value of the formatter is still a integer, it is converted to string. Suitable for human read represent. :param value: enumerate value :returns: a string represent the enumerate value

class namedstruct.**bitfield**(*basetype*, **properties*, ***arguments*)
bitfield are mini-struct with bit fields. It splits a integer primitive type like uint32 to several bit fields. The splitting is always performed with big-endian, which means the fields are ordered from the highest bits to lowest bits. Base type endian only affects the bit order when parsing the bytes to integer, but not the fields order. Unlike bit-fields in C/C++, the fields does not need to be aligned to byte boundary, and no padding between the fields unless defined explicitly. For example:

```
mybit = bitfield(uint64,
                 (4, 'first'),
                 (5, 'second'),
                 (2,),    # Padding bits
                 (19, 'third'),    # Can cross byte border
                 (1, 'array', 20), # A array of 20 1-bit numbers
                 name = 'mybit',
                 init = packvalue(2, 'second'),
                 extend = {'first': myenum, 'array': myenum2[20]})
```

__init__(*basetype*, **properties*, ***arguments*)
Initializer :param basetype: A integer primitive type to provide the bits

Parameters

- **properties** – placed arguments, definitions of fields. Each argument is a tuple, the first element is the bit-width, the second element is a field name. If a third element appears, the field is an bit-field array; if the second element does not appear, the field is some padding bits.
- **arguments** – keyword options

  name the type name

  init a initializer to initialize the struct

  extend similar to *extend* option in nstruct

  formatter similar to *formatter* option in nstruct

  prepack similar to *prepack* option in nstruct

class namedstruct.**optional**(*basetype*, *name*, *criteria*, *prepackfunc=None*)
Create a "optional" field in a struct. On unpacking, the field is only parsed when the specified criteria is met; on packing, the field is only packed when it appears (hasattr returns True). This function may also be done with sub-classing, but using *optional* is more convenient:

```
myopt = nstruct((uint16, 'data'),
                (uint8, 'hasextra'),
                (optional(uint32, 'extra', lambda x: x.hasextra),),
                name = 'myopt',
                prepack = packexpr(lambda x: hasattr(x, 'extra'), 'hasextra'))
```

- A optional type is placed in an anonymous field. In fact it creates an embedded struct with only one (optional) field.
- The criteria can only use fields that are defined before the optional field because the other fields are not yet parsed.
- Usually you should specify a *prepack* function to pack some identifier into the struct to identify that the optional field appears.
- The optional field does not exists when the struct is created with new(). Simply set a value to the attribute to create the optional field: myopt(data = 7, extra = 12)
- The optional type is a variable length type if and only if the *basetype* is a variable length type

__init__(*basetype*, *name*, *criteria*, *prepackfunc=None*)
Initializer.

Parameters

- **basetype** – the optional field type
- **name** – the name of the optional field
- **criteria** – a function to determine whether the optional field should be parsed.
- **prepackfunc** – function to execute before pack, like in nstruct

class namedstruct.**darray**(*innertype*, *name*, *size*, *padding=1*, *prepack=None*)
Create a dynamic array field in a struct. The length of the array is calculated by other fields of the struct. If the total size of the struct is stored, you should consider use *size* option and a variable length array (sometype[0]) as the last field, instead of calculating the array size yourself. If the array contains very simple elements like

primitives, it may be a better idea to calculate the total size of the struct from the array length and return the size from *size* option, because if the data is incomplete, the parsing quickly stops without the need to parse part of the array. Only use dynamic array when: the bytes size of the array cannot be determined, but the element size of the array is stored:

```
myopt = nstruct((uint16, 'data'),
                (uint8, 'extrasize'),
                (darray(mystruct, 'extras', lambda x: x.extrasize),),
                name = 'myopt',
                prepack = packexpr(lambda x: len(x.extras), 'hasextra'))
```

- A darray type is placed in an anonymous field. In fact it creates an embedded struct with only one field.
- The *size* function can only use fields that are defined before the optional field because the other fields are not yet parsed.
- Usually you should specify a *prepack* function to pack the array size into the struct
- *padding* option is available if it is necessary to pad the bytes size of the whole array to multiply of *padding*
- You can use *extend* with an array type (mystruct2[0]) to override the formatting of inner types or the whole array

__init__(*innertype*, *name*, *size*, *padding=1*, *prepack=None*)
Initializer.

Parameters

- **innertype** – type of array element
- **name** – field name
- **size** – a function to calculate the array length
- **padding** – align the array to padding-bytes boundary, like in nstruct
- **prepack** – prepack function, like in nstruct

class namedstruct.**nvariant**(*name*, *header=None*, *classifier=None*, *prepackfunc=None*, *padding=1*)
An *nvariant* struct is a specialized base for *nstruct*. Different from normal *nstruct*, it does not have *size* option, instead, its size is determined by subclassed structs.

A *variant* type can not be parsed with enough compatibility: if a new type of subclassed struct is not recognized, the whole data may be corrupted. It is not recommended to use this type for newly designed data structures, only use them to define data structures that: already exists; cannot be parsed by other ways.

__init__(*name*, *header=None*, *classifier=None*, *prepackfunc=None*, *padding=1*)
Initializer.

Parameters

- **name** – type name
- **header** – An embedded type, usually an nstruct. It is embedded to the nvariant and parsed before subclassing.
- **classifier** – same as *nstruct*
- **prepackfunc** – same as *nstruct*

namedstruct.**dump**(*val*, *humanread=True*, *dumpextra=False*, *typeinfo='flat'*)
Convert a parsed NamedStruct (probably with additional NamedStruct as fields) into a JSON-friendly format, with only Python primitives (dictionaries, lists, bytes, integers etc.) Then you may use json.dumps, or pprint to further process the result.

Parameters

- **val** – parsed result, may contain NamedStruct
- **humanread** – if True (default), convert raw data into readable format with type-defined formatters. For example, enumerators are converted into names, IP addresses are converted into dotted formats, etc.
- **dumpextra** – if True, dump "extra" data in '_extra' field. False (default) to ignore them.
- **typeinfo** – Add struct type information in the dump result. May be the following values:

  DUMPTYPE_FLAT ('flat') add a field '_type' for the type information (default)

  DUMPTYPE_KEY ('key') convert the value to dictionary like: {'<struc_type>': value}

  DUMPTYPE_NONE ('none') do not add type information

Returns "dump" format of val, suitable for JSON-encode or print.

2.2 Internal Classes

class namedstruct.namedstruct.**NamedStruct**(*parser*)

Store a binary struct message, which is serializable.

There are external interfaces for general usage as well as internal interfaces reserved for parser. See doc for each interface.

All the interface names start with _ to preserve normal names for fields.

__copy__()

Create a copy of the struct. It is always a deepcopy, in fact it packs the struct and unpack it again.

__deepcopy__(*memo*)

Create a copy of the struct.

__getstate__()

Try to pickle the struct. Register the typedef of the struct to make sure it can be pickled and transfered with the type information. Actually it transfers the packed bytes and the type name.

__init__(*parser*)

Constructor. Usually a NamedStruct is constructed automatically by a parser, you should not call the initializer yourself.

Parameters

- **parser** – parser to pack/unpack the struct
- **inlineparent** – if not None, this struct is embedded into another struct. An embedded struct acts like an agent to inlineparent - attributes are read from and stored into inlineparent.

__len__()

Get the struct size with padding. Usually this aligns a struct into 4-bytes or 8-bytes boundary.

Returns padded size of struct in bytes

__repr__(**args*, ***kwargs*)

Return the representation of the struct.

__setstate__(*state*)

Restore from pickled value.

__weakref__
list of weak references to the object (if defined)

_autosubclass()
Subclass a struct. When you modified some fields of a base class, you can create sub-classes with this method.

_create_embedded_indices()
Create indices for all the embedded structs. For parser internal use.

_extend(*newsub*)
Append a subclass (extension) after the base class. For parser internal use.

_get_embedded(*name*)
Return an embedded struct object to calculate the size or use _tobytes(True) to convert just the embedded parts.

Parameters **name** – either the original type, or the name of the original type. It is always the type used in type definitions, even if it is already replaced once or more.

Returns an embedded struct

_getextra()
Get the extra data of this struct.

_gettype()
Return current type of this struct

Returns a typedef object (e.g. nstruct)

_pack()
Pack current struct into bytes. For parser internal use.

Returns packed bytes

_prepack()
Prepack stage. For parser internal use.

_realsize()
Get the struct size without padding (or the "real size")

Returns the "real size" in bytes

static _registerPickleType(*name*, *typedef*)
Register a type with the specified name. After registration, NamedStruct with this type (and any sub-types) can be successfully pickled and transfered.

_replace_embedded_type(*name*, *newtype*)
Replace the embedded struct to a newly-created struct of another type (usually based on the original type). The attributes of the old struct is NOT preserved.

Parameters

- **name** – either the original type, or the name of the original type. It is always the type used in type definitions, even if it is already replaced once or more.
- **newtype** – the new type to replace

_setextra(*extradata*)
Set the _extra field in the struct, which stands for the additional ("extra") data after the defined fields.

_subclass(*parser*)
Create sub-classed struct from extra data, with specified parser. For parser internal use.

Parameters **parser** – parser of subclass

_tobytes(*skipprepack=False*)
Convert the struct to bytes. This is the standard way to convert a NamedStruct to bytes.

Parameters **skipprepack** – if True, the prepack stage is skipped. For parser internal use.

Returns converted bytes

_unpack(*data*)
Unpack a struct from bytes. For parser internal use.

_validate(*recursive=True*)
DEPRECATED structs are always unpacked now. _validate do nothing.

Force a unpack on this struct to check if there are any format errors. Sometimes a struct is not unpacked until attributes are read from it, if there are format errors in the original data, a BadFormatError is raised. Call _validate to ensure that the struct is fully well-formatted.

Parameters **recursive** – if True (default), also validate all sub-fields.

class namedstruct.namedstruct.**ParseError**
Base class for error in parsing

class namedstruct.namedstruct.**BadLenError**
Data size does not match struct length

class namedstruct.namedstruct.**BadFormatError**
Error in data format

2.3 Helper Functions

namedstruct.**sizefromlen**(*limit*, **properties*)
Factory to generate a function which get size from specified field with limits. Often used in nstruct "size" parameter.

To retrieve size without limit, simply use lambda expression: lambda x: x.header.length

Parameters

- **limit** – the maximum size limit, if the acquired value if larger then the limit, BadLenError is raised to protect against serious result like memory overflow or dead loop.
- **properties** – the name of the specified fields. Specify more than one string to form a property path, like: sizefromlen(256, 'header', 'length') -> s.header.length

Returns a function which takes a NamedStruct as parameter, and returns the length value from specified property path.

namedstruct.**packsize**(**properties*)
Revert to sizefromlen, store the struct size (len(struct)) to specified property path. The size includes padding. To store the size without padding, use packrealsize() instead. Often used in nstruct "prepack" parameter.

Parameters **properties** – specified field name, same as sizefromlen.

Returns a function which takes a NamedStruct as parameter, and pack the size to specified field.

namedstruct.**packrealsize**(**properties*)
Revert to sizefromlen, pack the struct real size (struct._realsize()) to specified property path. Unlike packsize, the size without padding is stored. Often used in nstruct "prepack" parameter.

Parameters **properties** – specified field name, same as sizefromlen.

Returns a function which takes a NamedStruct as parameter, and pack the size to specified field.

namedstruct.**packvalue**(*value*, **properties*)

Store a specified value to specified property path. Often used in nstruct "init" parameter.

Parameters

- **value** – a fixed value
- **properties** – specified field name, same as sizefromlen.

Returns a function which takes a NamedStruct as parameter, and store the value to property path.

namedstruct.**packexpr**(*func*, **properties*)

Store a evaluated value to specified property path. Often used in nstruct "prepack" parameter.

Parameters

- **func** – a function which takes a NamedStruct as parameter and returns a value, often a lambda expression
- **properties** – specified field name, same as sizefromlen.

Returns a function which takes a NamedStruct as parameter, and store the return value of func to property path.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Symbols

A

B

C

D

E

F

G

I

M

N

O

P

S

T

V