
CalPack Documentation

Release 0.0.3

KronoSKoderS

Nov 25, 2017

Contents

1	Examples	3
2	Index	5
2.1	models - a collection of classes and functions to create new custom packets.	5
3	Indices and tables	15
	Python Module Index	17

CalPack is the only package you'll need to create, generate and parse packets in an easy to use way. This module wraps the `cytpes` module into an easier to use interface and enabling more features specific to working with Packets.

CHAPTER 1

Examples

Creating a new packet is as simple as creating a python class:

```
from calpack import models

class UDP(models.Packet):
    source_port = models.IntField()
    dest_port = models.IntField()
    length = models.IntField()
    checksum = models.IntField()
```

Since calpak is a wrapper to ctypes, the above class is equivalent to the following ctypes.Structure:

```
import ctypes

class UDP(ctypes.Structure):
    _fields_ = [
        ('source_port', ctypes.c_uint64, 16),
        ('dest_port', ctypes.c_uint64, 16),
        ('length', ctypes.c_uint64, 16),
        ('checksum', ctypes.c_uint64, 16),
    ]
```

Interacting with the packet and it's field is also simple:

```
p = UDP()
p.source_port = 80
p.dest_port = 80
p.length = 8
```


2.1 models - a collection of classes and functions to create new custom packets.

This module is the building blocks for creating packets by using builtin and custom fields. It also provides the ability for users to create custom fields for their packets.

2.1.1 Packet Basics

In this section we cover the basics of how to create a packet and manipulate its contents.

Creating a Packet

Creating a custom packet requires inheriting the `Packet` class and then defining the Fields within the order they are expected to be seen:

```
from calpack import models

class Header(models.Packet):
    source = models.IntField()
    dest = models.IntField()
    data1 = models.IntField()
    data2 = models.IntField()
```

Once a packet is defined, creating an instance of that packet allows you to manipulate it:

```
my_pkt = Header()

my_pkt.source = 123
my_pkt.dest = 456
my_pkt.data1 = 789
```

```
print(my_pkt.source)
123
```

A packet can also be created with fields already populated:

```
my_pkt = Header(
    source = 1,
    dest = 2,
    data1 = 3,
    data2 = 4
)

print(my_pkt.source, my_pkt.dest, my_pkt.data1, my_pkt.data2)
(1, 2, 3, 4)
```

A packet can then be converted into a byte string:

```
my_pkt.to_bytes()
b' {\x00\xc8\x01\x15\x03\x00\x00'
```

In reverse, a packet can be created from a byte string array:

```
my_parsed_pkt = Header.from_bytes(b' {\x00\xc8\x01\x15\x03\x00\x00'})
print(my_parsed_pkt.source)
123

print(my_parsed_pkt.dest)
456

my_parsed_pkt == my_pkt
True

# Shows that the packets are two different objects
my_parsed_pkt is my_pkt
False
```

Packet fields can be easily copied from and/or compared to other packets of the same Packet subclass:

```
my_pkt2 = Header()
my_pkt2.source = my_pkt.source
my_pkt2.dest = 654

my_pkt.source == my_pkt2.source
True

my_pkt.dest == my_pkt2.dest
False
```

Packets themselves can also be compared:

```
my_pkt = Header()
my_pkt.source = 123
my_pkt.dest = 456
my_pkt.data1 = 789

my_pkt2 = Header()
my_pkt2.source = 123
```

```

my_pkt2.dest = 456
my_pkt2.data1 = 123

my_pkt == my_pkt2
False

my_pkt2.data1 = 789
my_pkt == my_pkt2
True

```

Note: Comparing two packets that are different classes but may have the same byte output will result in `False`

2.1.2 Advanced Packet Concepts

Creating simple packets with the basic `Fields` one thing but typically packets are more complex. For example, one might want to create a packet with an array of fields, or even encapsulating a packet within another as a field. This is easy to do within `calpack` through the use of the `ArrayField` or `PacketField` `Fields`.

Creating an Array of `IntField`'s

When deal with a lot of fields that are the same it can become a bear to create each field:

```

class my_long_packet(models.Packet):
    data1 = models.IntField()
    data2 = models.IntField()
    data3 = models.IntField()
    data4 = models.IntField()
    data5 = models.IntField()
    data6 = models.IntField()
    data7 = models.IntField()
    data8 = models.IntField()

```

This can be simplified by using the `models.ArrayField`:

```

class ArrayPacket(models.Packet):
    data = models.ArrayField(models.IntField(), 8)

```

Accessing the elements in the `ArrayField` is similar to that of a python list:

```

my_array_pkt = ArrayPacket()
my_array_pkt.data[0] = 123
print(my_array_pkt.data[0])
123

my_array_pkt.data = list(range(8))
print(my_array_pkt.data)
[0, 1, 2, 3, 4, 5, 6, 7]

for val in my_array_pkt.data:
    val = 100

print(my_array_pkt.data)
[100, 100, 100, 100, 100, 100, 100, 100]

```

Encapsulating another Packet within a Packet

Sometimes you might want to encapsulate another packet within a packet as a field. This can be done by using the `models.PacketField`:

```
class Header(models.Packet):
    source = models.IntField()
    destination = models.IntField()

class CustomPacket(models.Packet):
    header = models.PacketField(Header)
    spare = models.IntField()
    body = models.ArrayField(models.IntField(), 28)
```

Access to the fields within the encapsulated packet is as simple as calling that packets members:

```
pkt = CustomPacket()
pkt.header.source = 1
pkt.header.destination = 2
```

2.1.3 Packet Fields

calpack comes with some built-in `Field` classes that can be used right away.

IntField

The `IntField` is used to represent an integer. In the backend, this field uses the `ctypes.c_int64` or `ctypes.c_uint64` depending on whether the field is configured as signed or not. This is done by passing the `signed` parameter to the `IntField`:

```
int_field = models.IntField(signed=True)
```

Note: `IntField` is unsigned as a default.

Warning: If a signed value is set to an unsigned value (e.g any value less than 0) a `TypeError` will be raised.

Warning: although the example above defines a field outside of a `Packet`, this **cannot** be done in practice as each field within the packet **must** be a new instance of a `Field`.

If a specific bit length is desired, passing the `bit_len` parameter to the desired length:

```
int_field = models.IntField(bit_len=8)
```

Note: the default value for `bit_len` is 16

Warning: If `bit_len` is less than or equal 0 or greater than 64 a `ValueError` will be raised.

Warning: although the example above defines a field outside of a Packet, this **cannot** be done in practice as each field within the packet **must** be a new instance of a Field.

ArrayField

The `ArrayField` is used to create an array of fields. When creating the `ArrayField` two parameters must be passed:

1. An instance of the Field to be used
2. The size of the Array

Example:

```
array_field = models.ArrayField(
    # Note that this is an instance of the IntField
    models.IntField(bit_len=8, signed=True)
    12
)
```

Note: It's important to that the first argument is an **instance** of the Field and not the class

Warning: although the example above defines a field outside of a Packet, this **cannot** be done in practice as each field within the packet **must** be a new instance of a Field.

Interacting with the `ArrayField` is similar to that of a python list where `len` and individual member access can be done. The Field instance for the first parameter of the `ArrayField` can also be a `PacketField`:

```
class Point(models.Packet):
    x = models.IntField(bit_len=8)
    y = models.IntField(bit_len=8)

class ArrayPacket(models.Packet):
    points = models.ArrayField(
        models.PacketField(Point),
        8
    )

pkt = ArrayPacket()

for i, point in enumerate(pkt.points):
    point.x = i
    point.y = len(pkt.points) - 1

print([pkt.points[i].x, pkt.points[i].y for i in range(len(pkt.points))])
[(0, 8), (1, 7), (2, 6), (3, 5), (4, 4), (5, 3), (6, 2), (7, 1)]
```

Accessing the members of an `ArrayField` with a `PacketField` as the field type will be accessing instances of those packets:

```
class ArrayPacket (models.Packet) :
    points = array_field

pkt = ArrayPacket ()
pkt.points[0].x = 100
print (pkt.points[0].x)
100

print (pkt.points[0].y)
0 # default value of IntField
```

PacketField

The PacketField is used to encapsulate another already defined packet. The encapsulation of packets can be done multiple times as well:

```
class Point (models.Packet) :
    x = models.IntField(bit_len=8)
    y = models.IntField(bit_len=8)

class Rectangle (models.Packet) :
    top_left = models.PacketField(Point)
    top_right = models.PacketField(Point)
    bot_left = models.PacketField(Point)
    bot_right = models.PacketField(Point)

class TwoRectangles (models.Packet) :
    first_rect = models.PacketField(Rectangle)
    second_rect = models.PacketField(Rectangle)
```

Creating a Custom Field

Creating a custom Field is done by first subclassing the Field class from `calpack.models`. There are a few things to consider before creating a custom Field:

1. There are specific properties that **must** be defined within the class
2. Any methods defined must be a class method and cannot be used as an instance method
3. **You must have a basic understanding of the `ctypes` module specifically how to use the `ctypes.Structure` class.**

It is recommended that you study the way that [Structures](#) are created if you are looking to create a custom Field.

With that said, defining a custom Field is actually quite simple. The following properties **must** be defined at either the class level or in a custom `__init__`:

- `c_type` - a `ctypes` class to be used within the internal `ctypes.Structure` instance.
- `bit_len` - the length of the field in bits

Within your custom Field you can customize the following functions which are further defined in subsequent sections:

- `create_field_c_tuple`
- `py_to_c`

- `c_to_py`

`create_field_c_tuple`

This function is used to create the tuple that will go into the `_fields_` property of the Packet's internal `ctypes.Structure`. It's also important to note that the first elements in the tuple **must** be the property `self.field_name`. CalPack will automatically populate this property for the Field class.

As a default, CalPack's `Field` super class defines `create_field_c_tuple` as the following:

```
def create_field_c_tuple(self): return (self.field_name, ctypes.c_int)
```

However this can be customized to suit the needs of the custom field. Since this will be directly used to create the `ctypes.Structure._fields_` anything that is appropriate in creating the structure can be used here:

```
def create_field_c_tuple(self):
    # return a c_int with only 4 bits as a field tuple. Note: this is similar to how
    # we set the bit length of the IntField class.
    return (self.field_name, ctypes.c_int, 4)

def create_field_c_tuple(self):
    # return a c_int Array as a field tuple. Note: this is similar to how we set the
    # array size of the ArrayField class.
    return (self.field_name, ctypes.c_int * 10)
```

Warning: If the first element in the field tuple is **not** `self.field_name` then access to the internal c structure will be broken and the packets will not be accessible properly.

`py_to_c`

When setting a packet field to a value, that python object must be converted to a value that can be set for the internal `ctypes.Structure` object of the packet. Most occasions the value of the python object is already appropriate. By default this function does exactly that:

```
def py_to_c(self, val):
    return val
```

However in certain cases additional formatting, transformation or validation might be required. Use this function to override the behavior as needed.

`c_to_py`

Similar to going from python objects to c, the reverse of going from c to python might need to be configured properly. Most occasions the value of the c object is already appropriate. By default this function does exactly that:

```
def c_to_py(self, c_field):
    return c_field
```

However in certain cases additional formatting, transformation or validation might be required. Use this function to override the behavior as needed.

Full Example of Creating a Custom Field

The following is a quick example of how to create a custom Field within CalPack:

```

from calpack import models
import ctypes

class UInt30(models.Field):
    c_type = ctypes.c_uint32
    bit_len = 30

    def __init__(self):
        super(UInt30, self).__init__()
        self.max_size = (2 << self.bit_len) - 1

    def create_field_c_tuple(self):
        return (self.field_name, self.c_type, self.bit_len)

    def py_to_c(self, val):
        if val < 0:
            raise ValueError("UInt30 must be a positive number!")

        if val > self.max_size:
            raise ValueError("UInt30 cannot be greater than {}".format(self.max_size))

        return val

```

2.1.4 Class/Function specific Docs

class `models.Field` (*default_val=None*)

A Super class that all other fields inherit from. This class is NOT intended for direct use. Custom Fields MUST inherit from this class.

When creating a custom field you MUST define the `c_type` property with a valid `ctypes` data class.

c_to_py (*c_field*)

`c_to_py` - A function used to convert the `ctypes` object into a python object. As a default this function simply returns `c_field` directly from the `ctypes.Structure` object. It's up to the other `Field`'s to define this if further formatting is required in order to turn the `ctypes` value into something user friendly.

Parameters `c_field` - a `ctypes` object from the packet's internal `ctypes.Structure` object

create_field_c_tuple ()

`create_field_c_tuple` - A function used to create the required an field in the `ctypes.Structure._fields_` tuple. This must return a tuple that is acceptable for one of the items in the `_fields_` list of the `ctypes.Structure`.

The first value in the tuple MUST be `self.field_name` as this is used to access the internal `c` structure.

py_to_c (*val*)

`py_to_c` - A function used to convert a python object into a valid `ctypes` assignable object. As a default this function simply returns `val`. It's up to the other `Field`'s to define this if further formatting is required in order to set the internal structure of the packet.

Parameters `val` - the value the user is attempting to set the packet field to. This can be any python object.

class `models.IntField` (*bit_len=16, signed=False, default_val=0, little_endian=False*)

An Integer field. This field can be configured to be signed or unsigned. It's bit length can also be set, however the max bit length for this field is 64.

Parameters

- **bit_len** (*int*) – the length in bits of the integer. Max value of 64. (default 16)
- **signed** (*bool*) – whether to treat the int as a signed integer or unsigned integer (default unsigned)
- **default_val** (*int*) – the default value of the field (default 0)

Raises `ValueError` – if the `bit_len` is less than or equal to 0 or greater than 64

class `models.ArrayField` (*array_cls, array_size, default_val=None*)

A custom field for handling an array of fields

Parameters

- **array_cls** – a `calpack.models.Field` subclass **object** that represent the Field the array will be filled with.
- **array_size** (*int*) – the length of the array.

class `models.PacketField` (*packet_cls*)

A custom Field for handling another packet as a field.

Parameters `packet_cls` – A `calpack.models.Packet` subclass that represents another packet

class `models.Packet` (*c_pkt=None, **kwargs*)

A super class that custom packet classes MUST inherit from. This class is NOT intended to be used directly, but as a super class.

Example:

```
class Header(models.Packet):
    source = models.IntField()
    dest = models.IntField()
    data1 = models.IntField()
    data2 = models.IntField()
```

Parameters `c_pkt` – (Optional) a `ctypes.Structure` object that will be used at the internal c structure. This MUST have the same `_fields_` as the Packet would normally have in order for it to work properly.

`byte_size`

The byte size (assuming 8 bits to a byte) of the packet.

`c_pkt`

returns the internal c structure object being used

classmethod `from_bytes` (*buf*)

Creates a Packet from a bytes string

Parameters `buf` (*bytes*) – the bytes buffer that will be used to create the packet

Returns an Instance of the Packet as parsed from the bytes string

`num_words`

The number of words in the packet

`to_bytes()`

Converts the packet into a bytes string

Returns the packet as a byte string

Return type bytes

`models.typed_property(name, expected_type, default_val=None)`

Simple function used to ensure a specific type for a property defined within a class. This can ONLY be used within a class definition as the *self* keyword is used.

Parameters

- **name** (*str*) – the name of the variable. This can be anything, but cannot be already in use.
- **expected_type** (*type*) – the expected type. When setting this property at the class level, if the types do not match, a `TypeError` is raised.
- **default_val** – (Optional) the default value for the property. If not set, then `None` is used. This MUST be of the same type as *expected_type* or a `TypeError` is raised.

Returns the property

Raises `TypeError` – if the `default_val` or property's set value is not of type `expected_type`

CHAPTER 3

Indices and tables

- `genindex`
- `modindex`
- `search`

m

models, [12](#)

A

ArrayField (class in models), 13

B

byte_size (models.Packet attribute), 13

C

c_pkt (models.Packet attribute), 13

c_to_py() (models.Field method), 12

create_field_c_tuple() (models.Field method), 12

F

Field (class in models), 12

from_bytes() (models.Packet class method), 13

I

IntField (class in models), 12

M

models (module), 12

N

num_words (models.Packet attribute), 13

P

Packet (class in models), 13

PacketField (class in models), 13

py_to_c() (models.Field method), 12

T

to_bytes() (models.Packet method), 13

typed_property() (in module models), 14