CalPack Documentation

Release 0.0.3

KronoSKoderS

Feb 20, 2018

Contents

CalPack is a module that makes creating and parsing packets easy to do. This module wraps the `ctypes` module into an easier to use interface and enabling more features specific to working with Packets. Think of it as a way to "Transmogrify" your byte data into Packets and vice versa:

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_uint),
>>>         ('dest_port', ctypes.c_uint),
>>>         ('length', ctypes.c_uint),
>>>         ('checksum', ctypes.c_uint),
>>>     ]
```

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

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

CHAPTER 2

User Guide

If you've found yourself to be in a bind while using CalPack here is where you want to start. This is a set of guides on how to get and use CalPack.

CalPack User Guide

CHAPTER 3

CalPack Module Docs

If you're looking for detailed documenation for CalPack's classes and modules then look no further! You found it!

CalPack Documentation

CHAPTER 4

Contributing

If you're interested in contributing to CalPack here is where you can learn how

Contributing to CalPack

CHAPTER 5

Table of Contents

5.1 CalPack User Guide

This guide will help you get started on using CalPack. This guide is broken up into the following sections:

5.1.1 Introduction (A Brief History)

CalPack was created out of necessity for creating a way to parse custom packets in the space industry. These packets, typically stored in binary data files, would come from multiple unique interfaces (e.g. 1553) or common ones (e.g. UDP) with very custom data structures.

Naturally, `ctypes` was the first place I went to, but inspecting packet definitions using the syntax that `ctypes` used was confusing and not easy to use. Autocompletion of IDE's didn't pick up the field names and the reader had to have a basic knowledge of `ctypes`. And then CalPack was born.

But why CalPack? My first born is named Calvin and I wanted to name something after him as a tribute to him. Secondly, typing out `transmogrifier` is a nightmare. Finally, who doesn't like Calvin and Hobbes?

5.1.2 Installation

CalPack is hosted on pypi and can be installed by simply using `pypi`:

```
>>> pypi install CalPack
```

You can also install from the GitHub repo source:

```
>>> git clone https://github.com/KronoSKoderS/CalPack.git
>>> cd CalPack
>>> python setup.py install
```

5.1.3 Packet Basics

In this section we cover the basics of how to create a packet and manipulate it 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 UDP_Header(models.Packet):
...     source_port = models.IntField16()
...     dest_port = models.IntField16()
...     length = models.IntField16()
...     checksum = models.IntField16()
```

Note: The order in which the fields are defined is also the order in which the fields are set within the internal c structure.

If you desired to have a default value for a particular field, simply use the `default_val` param for the Field

```
>>> class UDP_Header(models.Packet):
...      source_port = models.IntField16(default_val=8888)
...      dest_port = models.IntField16(default_val=8000)
...      length = models.IntField16()
...      checksum = models.IntField16()
```

Upon creation of the Packet instance, any fields that haven't been set but have a default value will be automatically set to that default value.

Accessing and Manipulating the Fields

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

```
>>> my_pkt = UDP_Header()

>>> my_pkt.source_port = 8080
>>> my_pkt.dest_port = 8080
>>> my_pkt.length = 0x2
>>> my_pkt.checksum = 0x0

>>> print(my_pkt.source_port)
8080
```

An instance of a packet can also be created with fields already populated

```
>>> my_pkt = UDP_Header(
...      source_port=8080,
...      dest_port=8080,
...      length=0x2,
...      checksum=0x0
... )
```

(continues on next page)

(continued from previous page)

```
>>> print(my_pkt.source_port, my_pkt.dest_port, my_pkt.length, my_pkt.checksum)
8080 8080 2 0
```

Note: This is different than the `default_val` param. This value will overwrite that default value.

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

```
>>> my_pkt2 = UDP_Header()
>>> my_pkt2.source_port = my_pkt.source_port
>>> my_pkt2.dest_port = 8888

>>> my_pkt.source_port == my_pkt2.source_port
True

>>> my_pkt.dest_port == my_pkt2.dest_port
False
```

Packets themselves can also be compared

```
>>> my_pkt = UDP_Header()
>>> my_pkt.source_port = 123
>>> my_pkt.dest_port = 456
>>> my_pkt.length = 789

>>> my_pkt2 = UDP_Header()
>>> my_pkt2.source_port = 123
>>> my_pkt2.dest_port = 456
>>> my_pkt2.length = 123

>>> my_pkt == my_pkt2
False

>>> my_pkt2.length = 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`

Packets and Byte Strings

A packet instance can then be converted into a byte string

```
>>> my_pkt.to_bytes()
b'\x90\x1f\x90\x1f\x02\x00\x00\x00'
```

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

```
>>> my_parsed_pkt = UDP_Header.from_bytes(b'\x90\x1f\x90\x1f\x02\x00\x00\x00')
>>> print(my_parsed_pkt.source_port)
8080
```

(continues on next page)

(continued from previous page)

```
>>> print(my_parsed_pkt.dest_port)
8080

>>> my_parsed_pkt == my_pkt
True

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

5.1.4 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)
```

Writing to the array requires writing the entire list to the array

```
>>> my_array_pkt = ArrayPacket()
>>> my_array_pkt.data = list(range(8))
>>> my_array_pkt.data
(0, 1, 2, 3, 4, 5, 6, 7)
```

Access to individual member is currently readonly

```
>>> my_array_pkt.data[1] = 12
Traceback (most recent call last):
  File "<doctest adv_pkt[0]>", line 1, in <module>
    my_array_pkt.data[1] = 12
TypeError: 'tuple' object does not support item assignment

>>> print(my_array_pkt.data[1])
1
```

Iterating over the field using the for loop is also readonly as well.

```
>>> for val in my_array_pkt.data:
...     val = 100
>>> print(my_array_pkt.data)
(0, 1, 2, 3, 4, 5, 6, 7)

>>> for i, val in my_array_pkt.data:
...     my_array_pkt.data[i] = val * 2
Traceback (most recent call last):
  File "<doctest adv_pkt[0]>", line 1, in <module>
    my_array_pkt.data[1] = val * 2
TypeError: 'tuple' object does not support item assignment
```

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.source == 1
True
>>> pkt.header.destination = 2
>>> print(pkt.header.destination)
2
```

5.1.5 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_int` or `ctypes.c_uint` 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.

There are also four other `IntField` builtins that can be used to define a specific bit size:

- `IntField8` - wraps `c_uint8` or `c_int8`
- `IntField16` - wraps `c_uint16` or `c_int16`
- `IntField32` - wraps `c_uint32` or `c_int32`
- `IntField64` - wraps `c_uint64` or `c_int64`

Each one of these acts the same as `IntField` with the only difference being their bit size. 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 the size of the `ctypes` data object.

Warning: If bit_len is less than or equal 0 or greater than the size of the `ctypes` data object 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.

Unused Bits

When creating packets with the `IntField` and using the `bit_len` param, certain circumstances may lead to unexpected behavior, specifically with the sizing of the packet. This is due to the way that `ctypes` handles byte alignement. For example:

```
class simple(models.Packet):
    A = models.IntField8(bit_len=4)
    B = models.IntField8(bit_len=4)
    C = models.IntField8()
```

Will yield the following structure:

bit locations															
Byte 0								Byte 1							
0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
A	A	A	A	A	B	B	B	C	C	C	C	C	C	C	C

However, if not all bits of the Byte are not used, “spare” bits may be introduced:

```
class simple(models.Packet):
    A = models.IntField8(bit_len=2)
    B = models.IntField8(bit_len=4)
    C = models.IntField8()
```

bit locations															
Byte 0								Byte 1							
0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
A	A	B	B	B	B			C	C	C	C	C	C	C	C

Notice that the last two bits of the first Byte are empty and won't be used!

It is recommended that if you use the `bit_len` param for `IntField`, to make sure your field bits are byte aligned.

FloatField

The `FloatField` is used to represent a floating point number. Similar to the `IntField` there are three fields that represent floating points:

- `FloatField` - wraps `c_float`
- `DoubleField` - wraps `c_double`
- `LongDoubleField` - wraps `c_longdouble`

Each of these act the same, with the exception of their byte size.

BoolField

The `BoolField` is used to represent a `bool` object in python. This field wraps the `c_bool` data type in ctypes. The size of this field is one byte even though only `True` and `False` are the acceptable values.

FlagField

The `FlagField` is used to represent a "flag" in the packet. This is a single bit integer and can only store a value of 1 or 0. This field is similar to `IntField(bit_len=1)`.

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):
```

(continues on next page)

(continued from previous page)

```
    first_rect = models.PacketField(Rectangle)
    second_rect = models.PacketField(Rectangle)
```

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(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.IntField8()
    y = models.IntField8()

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
```

(continues on next page)

(continued from previous page)

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

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

ArrayField limitations and Workaround

When using ArrayFields, there are limitations of what types of Fields can be used. Any Field that returns a tuple size greater than 2 from the `create_field_c_tuple` method, cannot be used. An example Field is `IntField` with the `bit_len` set to a non byte aligned value. This is due to the complications of bit fields and arrays.

As a workaround to this, create a seperate Packet to be used as a PacketField within the Array. For Example:

```
from calpack import models

class simple_pkt(models.Packet):
    x = models.IntField8(bit_len=4)
    y = models.IntField8(bit_len=4)

class adv_pkt(models.Packet):
    simple_pkts = models.ArrayField(models.PacketField(simple_pkt), 8)
```

Note: is is recommended that you use byte aligned fields within the PacketField otherwise some bits might become unused.

5.1.6 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.

Field Function Overrides

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 classes.
    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
```

5.2 CalPack Documentation

5.2.1 Documentation Contents

models

A collection of classes and functions for creating custom packets.

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.

Parameters **default_val** – the default value of the field. This is set at instantiation of the Field

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 subclassesed `Field` 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=None*, *signed=False*, *default_val=0*)

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 `ctypes.sizeof(ctypes.c_int) * 8`. This wraps around the `ctypes.c_int` or `ctypes.c_uint` data type.

Warning: A word of caution when using the `bit_len`. If the combination of IntFields with the bit_len set are not byte aligned, there is the possibility of "spare" bits not accessible but used in the overall strucuture. See *Unused Bits* for more information

Parameters

- **bit_len** (*int*) – the length in bits of the integer.
- **signed** (*bool*) – whether to treat the int as an 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 `ctypes.sizeof(ctypes.c_int) * 8`

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 subclassesed `Field` 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.**IntField8**(*bit_len=None*, *signed=False*, *default_val=0*)

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 8. This wraps around the `ctypes.c_int8` or `ctypes.c_uint8` data type.

> **Warning:** A word of caution when using the `bit_len`. If the combination of IntFields with the bit_len set are not byte aligned, there is the possibility of "spare" bits not accessible but used in the overall strucuture. See *Unused Bits* for more information

Parameters

- **bit_len** (*int*) – the length in bits of the integer.
- **signed** (*bool*) – whether to treat the int as an 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 8

class models.**IntField16**(*bit_len=None*, *signed=False*, *default_val=0*)

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 16. This wraps around the `ctypes.c_int16` or `ctypes.c_uint16` data type.

> **Warning:** A word of caution when using the `bit_len`. If the combination of IntFields with the bit_len set are not byte aligned, there is the possibility of "spare" bits not accessible but used in the overall strucuture. See *Unused Bits* for more information

Parameters

- **bit_len** (*int*) – the length in bits of the integer.
- **signed** (*bool*) – whether to treat the int as an 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 16

class models.**IntField32**(*bit_len=None*, *signed=False*, *default_val=0*)

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 32. This wraps around the `ctypes.c_int32` or `ctypes.c_uint32` data type.

> **Warning:** A word of caution when using the `bit_len`. If the combination of IntFields with the bit_len set are not byte aligned, there is the possibility of "spare" bits not accessible but used in the overall strucuture. See *Unused Bits* for more information

Parameters

- **bit_len** (*int*) – the length in bits of the integer.
- **signed** (*bool*) – whether to treat the int as an 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 32

class models.**IntField64**(*bit_len=None*, *signed=False*, *default_val=0*)

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. This wraps around the `ctypes.c_int64` or `ctypes.c_uint64` data type.

> **Warning:** A word of caution when using the `bit_len`. If the combination of IntFields with the bit_len set are not byte aligned, there is the possibility of "spare" bits not accessible but used in the overall strucuture. See *Unused Bits* for more information

Parameters

- **bit_len** (*int*) – the length in bits of the integer.
- **signed** (*bool*) – whether to treat the int as an 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. Only tuples or other ArrayFields can be written to the

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.

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

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 subclassesed `Field` 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.**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

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 subclassesed `Field` 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.**FlagField**(*default_val=False*)
A custom field for handling single bit 'flags'.

Parameters **default_val** (*bool*) – the default value of the field (default False)

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

c_type
alias of `ctypes.c_ubyte`

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 subclassesed `Field` 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.**FloatField**(*default_val=0.0*)
A custom field for handling floating point numbers.

c_type
alias of `ctypes.c_float`

class models.**DoubleField**(*default_val=0.0*)
A custom field for handling double floating point numbers

c_type
alias of `ctypes.c_double`

class models.**LongDoubleField**(*default_val=0.0*)
A custom field for handling long double floating point numbers

c_type
alias of `ctypes.c_longdouble`

class models.**BoolField**(*default_val=False*)
A custom field for handling Boolean types

c_type
alias of `ctypes.c_bool`

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 subclassesed `Field` 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.**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.

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

get_c_field(*field_name*)
gets the value of the field value of the internal c structure. :param str field_name: the name of the field to get :returns: the field value

set_c_field(*field_name*, *val*)
sets the value of the internal c structure.

Parameters

- **field_name** (*str*) – the name of the field to set
- **val** – a cytpes compatible value to set the field to

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`

5.3 Contributing to CalPack

Welcome! And thanks for being interested in working on CalPack. Before you start "hammering" away at the code, please read through *How to Contribute*. If you have any questions feel free to ask any of the core developers and we'll gladly help you.

5.3.1 How to Contribute

Contributing to CalPack can come in many forms including:

- Testing
- Documentation
- Developing New Features and/or Fixing Known issues
- Reporting Issues

Don't feel like you need a specific amount of experience to contribute. Simply reporting an issue is contributing! In order to develop *some* of the features within CalPack some knowledge of `ctypes` is needed.

GitHub, Issue Tracking and Branching

Development of CalPack is done within GitHub. Any issues that are encountered or reported are tracked within GitHub's internal issues tracking.

When developing code we use the following 3 major branches:

- `prod` - production level code. This is where relases are tracked. It's from this version that the pypi repo is updated. Only `integ` branch can be pushed to this branch. This branch requires admin approval for Pull Requests to be merged.
- `integ` - integration level code. This is *nearly* a clone of `prod`. The `dev` branch can push to this branch as well any hotfixes (small critical changes that need to be deployed immediately.
- `dev` - Main development happens here. Any Pull Requests from developers should be against this branch.

Any new Pull Request should be against the `dev` branch. When working on specific features, branch from the `dev` branch using the branch name `feature/<feature-topic>`. When a critical issue is identified that needs to be fixed immediately, branch from the `integ` branch using the branch name `hotfix/<hotfix-topic>`

When we do releases, brances from the `integ` branch will be done using the branch name `v<version number>`.

After each push into any branch a Travis CI and Appveyor to run tests on differing python versions on linux and Windows. After successfully passing the unittests coverage results are uploaded to coveralls and codacy.

Warning: Before a Pull Request into the `prod` branch, all unittests and QA checks from Coveralls and Codacy have to pass first.

Sprints and Sprint Planning

While issues are tracked within GitHub, we additionally use ZenHub for prioritzing and planning future development of CalPack. For instant message communication we use slack. To be invited to the channel send a message to `superuser<dot>kronos<at>gmail<dot>com`.

Sprints typically run for 2 weeks at a time.

Developing code

The best way to start developing is to look through the issues listed in the issues page of GitHub. When creating new features or changes that affect the code, it's imperative that unittests are updated as well. This may require the creation of new unittests. Any new tests that are implemented or old tests that have changed, need to go through a review with at least another CalPack developer.

We try to use a TDD approach to development of new features. Tests should be written **first** and then the features implemented. However this isn't always necessary. If you decide to create a new feature or make changes to the code, please add any additional tests that will ensure the quality of the code you've created.

Documentation

Documentation is done using Sphinx and reStructured Text.

Testing

Even though developers should be creating unittests along with their code, additional testing may be required to ensure that the overal functionality is preserved. Additional tests are always welcome so long as they provide value added to the testing of the code. Duplicate testing should be avoided and will be rejected during review if found.

5.3.2 Suggested Tools

If you're a developer, you probbaly have a set of tools you are used to using. If you're new you're probaly still exploring. Here are a few of my favorite tools that I use for development on CalPack;

- Visual Studio Code - multi-platform editor with python support including auto-complete and pylint.
- Anaconda python - Python distribution with built-in libraries already installed. I also prefer its environment creation over pipenv

5.3.3 Contributor Covenant Code of Conduct

Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, gender identity and expression, level of experience, nationality, personal appearance, race, religion, or sexual identity and orientation.

Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at `superuser<dot>kronos<at>gmail<dot>com`. The project team will review and investigate all complaints, and will respond in a way that it deems appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

Attribution

This Code of Conduct is adapted from the Contributor Covenant , version 1.4, available here

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

Index