pycstruct Documentation

Release 0.12.1

Joel Midstjärna

Apr 28, 2023

Contents

pycstruct is a python library for converting binary data to and from ordinary python dictionaries or specific instance objects.

Data is defined similar to what is done in C language structs, unions, bitfields and enums.

Typical usage of this library is read/write binary files or binary data transmitted over a network.

Following complex C types are supported:

- Structs
- Unions
- Bitfields
- Enums

These types may consist of any traditional data types (integer, unsigned integer, boolean and float) between 1 to 8 bytes large, arrays (lists), and strings (ASCII/UTF-8).

Structs, unions, bitfields and enums can be embedded inside other structs/unions in any level.

Individual elements can be stored / read in any byte order and alignment.

pycstruct also supports parsing of existing C language source code to automatically generate the pycstruct definitions / instances.

CHAPTER 1

Installation

1.1 Python version compatibility

pycstruct is only compatible with python version 3.6 or later.

1.2 Basic Installation

Install pycstruct with **pip**:

```
python3 -m pip install pycstruct
```

1.3 Extra dependencies to parse source code

To be able to parse C source code (optional) you also need to install castxml.

castxml supports all the major platforms such as Linux, Windows and OS X.

CHAPTER 2

Overview

2.1 Background

Typically python applications don't care about memory layout of the used variables or objects. This is generally not a problem when parsing text based data such as JSON, XML data. However, when parsing binary data the Python language and standard library has limited support for this.

The pycstruct library solves this problem by allowing the user to define the memory layout of an "object". Once the memory layout has been defined data can serialized or deserialized into/from simple python dictionaries or specific instance objects.

2.2 Why and when does the memory layout matter?

Strict memory layout is required when reading and writing binary data, such as:

- Binary file formats
- Binary network data

2.3 Structs

Memory layout of an object is defined using the *pycstruct.StructDef()* object. For example:

```
myStruct = pycstruct.StructDef()
myStruct.add('int8', 'mySmallInteger')
myStruct.add('uint32', 'myUnsignedInteger')
myStruct.add('float32', 'myFloatingPointNumber')
```

The above example corresponds to following layout:

Size in bytes	Type	Name
1	Signed integer	mySmallInteger
4	Unsigned integer	myUnsignedInteger
4	Floating point number	myFloatingPointNumber

Now, when the layout has been defined, you can write binary data using ordinary python dictionaries.

```
myDict = {}
myDict['mySmallInteger'] = -4
myDict['myUnsignedInteger'] = 12345
myDict['myFloatingPointNumber'] = 3.1415

myByteArray = myStruct.serialize(myDict)
```

myByteArray is now a byte array that can for example can be written to a file or transmitted over a network.

The reverse process looks like this (assuming data is stored in the file myDataFile.dat):

```
with open('myDataFile.dat', 'rb') as f:
    inbytes = f.read()

myDict2 = myStruct.deserialize(inbytes)
```

myDict2 will now be a dictionary with the fields mySmallInteger, myUnsignedInteger and myFloatingPointNumber.

2.4 Arrays

Arrays are added like this:

```
myStruct = pycstruct.StructDef()
myStruct.add('int32', 'myArray', shape=100)
```

Now myArray will be an array with 100 elements.

```
myDict = {}
myDict['myArray'] = [32, 11]

myByteArray = myStruct.serialize(myDict)
```

Note that you don't have to provide all elements of the array in the dictionary. Elements not defined will be set to 0 during serialization.

2.5 Ndim arrays

The shape can be a tuple for multi dimensional arrays. The last element of the tuple is the fastest dimension.

```
myStruct = pycstruct.StructDef()
myStruct.add('int32', 'myNdimArray', shape=(100, 50, 2))
```

2.6 Strings

Strings are always encoded as UTF-8. UTF-8 is backwards compatible with ASCII, thus ASCII strings are also supported.

```
myStruct = pycstruct.StructDef()
myStruct.add('utf-8', 'myString', length=50)
```

Now myString will be a string of 50 bytes. Note that:

- Non-ASCII characters are larger than one byte. Thus the number of characters might not be equal to the specified length (which is in bytes not characters)
- The last byte is used as null-termination and should not be used for characters data.

To write a string:

```
myDict = {}
myDict['myString'] = "this is a string"

myByteArray = myStruct.serialize(myDict)
```

If you need another encoding that UTF-8 or ASCII it is recommended that you define your element as an array of uint8. Then you can decode/encode the array to any format you want.

2.7 Embedding Structs

Embedding structs in other structs is simple:

```
myChildStruct = pycstruct.StructDef()
myChildStruct.add('int8', 'myChildInteger')

myParentStruct = pycstruct.StructDef()
myParentStruct.add('int8', 'myParentInteger')
myParentStruct.add(myChildStruct, 'myChild')
```

Now myParentStruct includes myChildStruct.

```
myChildDict = {}
myChildDict['myChildInteger'] = 7

myParentDict['myParentInteger'] = 45
myParentDict['myChild'] = myChildDict

myByteArray = myStruct.serialize(myParentDict)
```

Note that you can also make an array of child structs by setting the length argument when adding the element.

2.8 Unions

Unions are defined using the *pycstruct.StructDef()* class, but the union argument in the construct shall be set to True.

When deserializing a binary for a union, pycstruct tries to generate a dictionary for each member. If any of the members fails due to formatting errors these members will be ignored.

When serializing a dictionary into a binary pycstruct will just pick the first member it finds in the dictionary. Therefore you should only define the member that you which to serialize in your dictionary.

2.9 Bitfields

The struct definition requires that the size of each member is 1, 2, 4 or 8 bytes. *pycstruct.BitfieldDef()* allows you to define members that have any size between 1 to 64 bits.

```
myBitfield = pycstruct.BitfieldDef()

myBitfield.add("myBit",1)
myBitfield.add("myTwoBits",2)
myBitfield.add("myFourSignedBits",4 ,signed=True)
```

The above bitfield will allocate one byte with following layout:

BIT index 7	BIT index 6 - 3	BIT index 2-1	BIT index 0
Unused	MyFourSignedBits	myTwoBits	myBit

To add myBitfield to a struct def:

```
myStruct = pycstruct.StructDef()
myStruct.add(myBitfield, 'myBitfieldChild')
```

To access myBitfield

```
myBitfieldDict = {}
myBitfieldDict['myBit'] = 0
myBitfieldDict['myTwoBit'] = 3
myBitfieldDict['myFourSignedBits'] = -1

myDict = {}
myDict['myBitfieldChild'] = myBitfieldDict

myByteArray = myStruct.serialize(myDict)
```

2.10 Enum

pycstruct.EnumDef() allows your to define a signed integer of size 1, 2, 3, ... or 8 bytes with a defined set of values (constants):

```
myEnum = pycstruct.EnumDef()

myEnum.add('myConstantM3',-3)
myEnum.add('myConstant0',0)
myEnum.add('myConstant5',5)
myEnum.add('myConstant44',44)
```

To add an enum to a struct:

```
myStruct = pycstruct.StructDef()
myStruct.add(myEnum, 'myEnumChild')
```

The constants are accessed as strings:

```
myDict = {}
myDict['myEnumChild'] = 'myConstant5'

myByteArray = myStruct.serialize(myDict)
```

Setting myEnumChild to a value not defined in the EnumDef will result in an exception.

2.11 Byte order

Structs, bitfields and enums are by default read and written in the native byte order. However, you can always override the default byteorder by providing the byteorder argument.

```
myStruct = pycstruct.StructDef(default_byteorder = 'big')
myStruct.add('int16', 'willBeBigEndian')
myStruct.add('int32', 'willBeBigEndianAlso')
myStruct.add('int32', 'willBeLittleEndian', byteorder = 'little')

myBitfield = pycstruct.BitfieldDef(byteorder = 'little')

myEnum = pycstruct.EnumDef(byteorder = 'big')
```

2.12 Alignment and padding

Compilers usually add padding in-between elements in structs to secure individual elements are put on addresses that can be accessed efficiently. Also, padding is added in the end of the structs when required so that an array of the struct can be made without "memory gaps".

Padding depends on the alignment of the CPU architecture (typically 32 or 64 bits on modern architectures), the size of individual items in the struct and the position of the items in the struct.

The padding behavior can be removed by most compilers, usually adding a compiler flag or directive such as:

```
#pragma pack(1)
```

pycstruct is by default not adding any padding, i.e. the structs are packed. However by providing the alignment argument padding will be added automatically.

```
noPadding_Default          = pycstruct.StructDef(alignment = 1)
paddedFor16BitArchitecture = pycstruct.StructDef(alignment = 2)
paddedFor32BitArchitecture = pycstruct.StructDef(alignment = 4)
paddedFor64BitArchitecture = pycstruct.StructDef(alignment = 8)
```

Lets add padding to the first example in this overview:

```
myStruct = pycstruct.StructDef(alignment = 8)
myStruct.add('int8', 'mySmallInteger')
myStruct.add('uint32', 'myUnsignedInteger')
myStruct.add('float32', 'myFloatingPointNumber')
```

The above example will now have following layout:

Size in bytes	Type	Name
1	Signed integer	mySmallInteger
1	Unsigned integer	__pad_0[0]
1	Unsigned integer	__pad_0[1]
1	Unsigned integer	__pad_0[2]
4	Unsigned integer	myUnsignedInteger
4	Floating point number	myFloatingPointNumber

Note that when parsing source code, pycstruct has some limitations regarding padding of bitfields. See *Limitations*.

2.13 Parsing source code

Instead of manually creating the definitions as described above, C source code files can be parsed and the definitions will be generated automatically with *pycstruct.parse_file()*.

It is also possible to write the source code into a string and parse it with *pycstruct.parse_str()*.

Internally pycstruct use the external tool castxml which needs to be installed and put in the current path.

2.14 Instance objects

Most examples in this section are using dictionaries. An alternative of using dictionaries to represent the object is to use *pycstruct.Instance()* objects.

Instance objects has following advantages over dictionaries:

- Data is only serialized/deserialized when accessed
- Data is validated for each element/attribute access. I.e. you will get an exception if you try to set an element/attribute to a value that is not supported by the definition.
- Data is accessed by attribute name instead of key indexing

Instance objects are created from the *pycstruct.StructDef()* or *pycstruct.BitfieldDef()* object.

```
myStruct = pycstruct.StructDef()
#.... Add some elements to myStruct here
instanceOfMyStruct = myStruct.instance()

myBitfield = pycstruct.BitfieldDef()
#.... Add some elements to myBitfield here
instanceOfMyBitfield = myBitfield.instance()
```

2.15 Deserialize with numpy

The structure definitions can be used together with numpy, with some restrictions.

This provides an easy way to describe complex numpy dtype, especially compound dtypes.

There is some restructions:

- bitfields and enums are not supported
- strings are not decoded (that's still bytes)

This can be used for use cases requiring very fast processing, or smart indexing.

The structure definitions provides a method *dtype* which can be read by numpy.

```
import pycstruct
import numpy

# Define a RGBA color
color_t = pycstruct.StructDef()
color_t.add("uint8", "r")
color_t.add("uint8", "g")
color_t.add("uint8", "b")
color_t.add("uint8", "a")

# Define a vector of RGBA
colorarray_t = pycstruct.StructDef()
colorarray_t.add(color_t, "vector", shape=200)

# Dummy data
raw = b"\x20\x30\x40\xFF" * 200

# Deserialize the raw bytes
colorarray = numpy.frombuffer(raw, dtype=colorarray_t.dtype(), count=1)
# numpy.frombuffer deserialize arrays. In this case there is
# a single element of colorarray_t, which can be unstacked
colorarray = colorarray[0]

# Elements can be accessed by names
# Here we can access to the whole red components is a single request
red_component = colorarray["vector"]["r"]
assert red_component.dtype == numpy.uint8
assert red_component.shape == (200, )
```

Numpy also provides record array which can be used like the instance objects.

```
colorarray = numpy.frombuffer(raw, dtype=colorarray_t.dtype())[0]

# Create a record array
colorarray = numpy.rec.array(colorarray)

# Elements can be accessed by attributes
assert colorarray.vector.r.dtype == numpy.uint8
assert colorarray.vector.r.shape == (200, )
```

CHAPTER 3

Examples

3.1 Simple Example

Following C has a structure (person) with a set of elements that are written to a binary file.

```
#include <stdio.h>
#include <stdbool.h>
#include <string.h>

#pragma pack(1) // To secure no padding is added in struct

struct person
{
    char name[50];
    unsigned int age;
    float height;
    bool is_male;
    unsigned int nbr_of_children;
    unsigned int child_ages[10];
};


void main(void) {
    struct person p;
    memset(&p, 0, sizeof(struct person));

    strcpy(p.name, "Foo Bar");
    p.age = 42;
    p.height = 1.75; // m
    p.is_male = true;
    p.nbr_of_children = 2;
    p.child_ages[0] = 7;
    p.child_ages[1] = 9;
```

(continues on next page)

(continued from previous page)

```
    FILE *f = fopen("simple_example.dat", "w");
    fwrite(&p, sizeof(struct person), 1, f);
    fclose(f);
}
```

To read the binary file using pycstruct following code required.

```
import pycstruct

person = pycstruct.StructDef()
person.add('utf-8', 'name', length=50)
person.add('uint32', 'age')
person.add('float32','height')
person.add('bool8', 'is_male')
person.add('uint32', 'nbr_of_children')
person.add('uint32', 'child_ages', length=10)

with open('simple_example.dat', 'rb') as f:
    inbytes = f.read()

# Dictionary representation
result = person.deserialize(inbytes)
print('Dictionary object:')
print(str(result))

# Alternative, Instance representation
instance = person.instance(inbytes)
print('\nInstance object:')
print(f'name: {instance.name}')
print(f'nbr_of_children: {instance.nbr_of_children}')
print(f'child_ages[1]: {instance.child_ages[1]}')
```

The produced output will be:

```
Dictionary object:
{'name': 'Foo Bar', 'is_male': True, 'nbr_of_children': 2,
 'age': 42, 'child_ages': [7, 9, 0, 0, 0, 0, 0, 0, 0, 0],
 'height': 1.75}

Instance object:
name: Foo Bar
nbr_of_children: 2
child_ages[1]: 9
```

To write a binary file from python using the same structure using pycstruct following code is required.

```
import pycstruct

person = pycstruct.StructDef()
person.add('utf-8', 'name', length=50)
person.add('uint32', 'age')
person.add('float32','height')
person.add('bool8', 'is_male')
person.add('uint32', 'nbr_of_children')
person.add('uint32', 'child_ages', length=10)

# Dictionary representation
```

(continues on next page)

(continued from previous page)

```
mrGreen = {}
mrGreen['name'] = "MR Green"
mrGreen['age'] = 50
mrGreen['height'] = 1.93
mrGreen['is_male'] = True
mrGreen['nbr_of_children'] = 3
mrGreen['child_ages'] = [13,24,12]
buffer = person.serialize(mrGreen)

# Alternative, Instance representation
mrGreen = person.instance()
mrGreen.name = "MR Green"
mrGreen.age = 50
mrGreen.height = 1.93
mrGreen.is_male = True
mrGreen.nbr_of_children = 3
mrGreen.child_ages[0] = 13
mrGreen.child_ages[1] = 24
mrGreen.child_ages[2] = 12
buffer = bytes(mrGreen)

# Write to file
f = open('simple_example_mr_green.dat','wb')
f.write(buffer)
f.close()
```

3.2 Embedded Struct Example

A struct can also include another struct.

Following C structure:

```
struct car_s
{
    unsigned short year;
    char model[50];
    char registration_number[10];
};

struct garage_s
{
    struct car_s cars[20];
    unsigned char nbr_registered_parkings;
};

struct house_s {
    unsigned char nbr_of_levels;
    struct garage_s garage;
};
```

Is defined as following:

```
car = pycstruct.StructDef()
car.add('uint16', 'year')
```

(continues on next page)

(continued from previous page)

```
car.add('utf-8', 'model', length=50)
car.add('utf-8', 'registration_number', length=10)

garage = pycstruct.StructDef()
garage.add(car, 'cars', length=20)
garage.add('uint8', 'nbr_registered_parkings')

house = pycstruct.StructDef()
house.add('uint8', 'nbr_of_levels')
house.add(garage, 'garage')
```

To print the model number of the first car:

```
# Dictionary representation
my_house = house.deserialize(databuffer)
print(my_house['garage']['cars'][0]['model'])

# Alternative, Instance representation
my_house = house.instance(databuffer)
print(my_house.garage.cars[0].model)
```

3.3 Parsing source code

Assume the C code listed in the first example is named simple_example.c. Then you could parse the source code instead of manually creating the definitions:

```
import pycstruct

definitions = pycstruct.parse_file('simple_example.c')

with open('simple_example.dat', 'rb') as f:
    inbytes = f.read()

# Dictionary representation
result = definitions['person'].deserialize(inbytes)
print(str(result))

# Alternative, Instance representation
instance = definitions['person'].instance(inbytes)
```

The produced output will be the same is in the first example.

3.4 Parsing source code strings

You can also define the source in a string and parse it:

```
c_str = '''
struct a_struct {
  int member_a;
  int member_b;
};
struct a_bitfield {
```

(continues on next page)

(continued from previous page)

```
  unsigned int bmem_a : 3;
  unsigned int bmem_b : 1;
};
'''

result = pycstruct.parse_str(c_str)
```

In the above example result['a_struct'] will be a pcstruct.StructDef instance and result['a_bitfield'] will be a pycstruct.BitfieldDef instance.

CHAPTER 4

Frequently Asked Questions

4.1 How to disable printouts from pycstruct

pycstruct utilizes the standard logging module for printing out warnings. To disable logging from the pycstruct module add following to your application

```
import logging
logging.getLogger('pycstruct').setLevel(logging.CRITICAL)
```

CHAPTER 5

Limitations

5.1 Padding of bitfields

How bitfields are stored in memory is not defined by the C standard. pycstruct c parser tries to keep consecutive defined bitfields together, however gcc (for example) will split bitfields when alignment is needed.

Following example will be a problem:

```
struct a_struct {
  char m1;        // = 8 bits
  int bf1 : 2;    // = 2 bits
  int bf2 : 23;   // = 23 bits
};
```

pycstruct c parser will layout the struct like this using 32 bit alignment:

Size in bytes	Type	Name
1	Unsigned integer	m1
1	Unsigned integer	__pad_0[0]
1	Unsigned integer	__pad_0[1]
1	Unsigned integer	__pad_0[2]
4	Bitfield	bf1 + bf2

gcc will layout the struct like this using 32 bit alignment:

Size in bytes	Type	Name
1	Unsigned integer	m1
1	Bitfield	bf1
1	Unsigned integer	__pad_0[0]
1	Unsigned integer	__pad_0[1]
3	Bitfield	bf2

One workaround is to pack the struct using compiler directives.

Another workaround is to manually add padding bytes in the struct in-between the bitfield definitions.

CHAPTER 6

Reference

6.1 StructDef (struct and union representation)

class pycstruct.**StructDef**(*default_byteorder='native'*, *alignment=1*, *union=False*)

This class represents a struct or a union definition

Parameters

- **default_byteorder** (*str, optional*) – Byte order of each element unless explicitly set for the element. Valid values are 'native', 'little' and 'big'.
- **alignment** (*str, optional*) – Alignment of elements in bytes. If set to a value > 1 padding will be added between elements when necessary. Use 4 for 32 bit architectures, 8 for 64 bit architectures unless packing is performed.
- **union** (*boolean, optional*) – If this is set the True, the instance will behave like a union instead of a struct, i.e. all elements share the same data (same start address). Default is False.

add(*datatype*, *name*, *length=1*, *byteorder=''*, *same_level=False*, *shape=None*)

Add a new element in the struct/union definition. The element will be added directly after the previous element if a struct or in parallel with the previous element if union. Padding might be added depending on the alignment setting.

- Supported data types:

Name	Size in bytes	Comment
int8	1	Integer
uint8	1	Unsigned integer
bool8	1	True (<>0) or False (0)
int16	2	Integer
uint16	2	Unsigned integer
bool16	2	True (<>0) or False (0)
float16	2	Floating point number
int32	4	Integer
uint32	4	Unsigned integer
bool32	4	True (<>0) or False (0)
float32	4	Floating point number
int64	8	Integer
uint64	8	Unsigned integer
bool64	8	True (<>0) or False (0)
float64	8	Floating point number
utf-8	1	UTF-8/ASCII string. Use length parameter to set the length of the string including null termination
struct	struct size	Embedded struct. The actual StructDef object shall be set as type and not 'struct' string.
bit-field	bitfield size	Bitfield. The actual *BitfieldDef()* object shall be set as type and not 'bitfield' string.
enum	enum size	Enum. The actual *EnumDef()* object shall be set as type and not 'enum' string.

Parameters

- **datatype** (*str | _BaseDef*) – Element data type. See above.
- **name** (*str*) – Name of element. Needs to be unique.
- **length** (*int, optional*) – Number of elements. If > 1 this is an array/list of elements with equal size. Default is 1. This should only be specified for string size. Use *shape* for arrays.
- **shape** (*int, tuple, optional*) – If specified an array of this shape is defined. It supported, int, and tuple of int for multi-dimentional arrays (the last is the fast axis)
- **byteorder** (*str, optional*) – Byteorder of this element. Valid values are 'native', 'little' and 'big'. If not specified the default byteorder is used.
- **same_level** (*bool, optional*) – Relevant if adding embedded bitfield. If True, the serialized or deserialized dictionary keys will be on the same level as the parent. Default is False.

create_empty_data()
Create an empty dictionary with all keys

Returns A dictionary keyed with the element names. Values are "empty" or 0.

Return type dict

deserialize(*buffer*, *offset=0*)
Deserialize buffer into dictionary

dtype()
Returns the dtype of this structure as defined by numpy.

This allows to use the pycstruct modelization together with numpy to read C structures from buffers.

```
color_t = StructDef()
color_t.add("uint8", "r")
color_t.add("uint8", "g")
color_t.add("uint8", "b")
color_t.add("uint8", "a")
raw = b""
color = numpy.frombuffer(raw, dtype=color_t.dtype())
```

Returns a python dict representing a numpy dtype

Return type dict

get_field_type(*name*)
Returns the type of a field of this struct.

Returns Type if the field

Return type _BaseDef

instance(*buffer=None*, *buffer_offset=0*)
Create an instance of this struct / union.

This is an alternative of using dictionaries and the *StructDef.serialize()*/ *StructDef.deserialize()* methods for representing the data.

Parameters

- **buffer** (*bytearray, optional*) – Byte buffer where data is stored. If no buffer is provided a new byte buffer will be created and the instance will be 'empty'.
- **buffer_offset** (*int, optional*) – Start offset in the buffer. This means that you can have multiple Instances (or other data) that shares the same buffer.

Returns A new Instance object

Return type *Instance()*

remove_from(*name*)
Remove all elements from a specific element

This function is useful to create a sub-set of a struct.

param name Name of element to remove and all after this element

type name str

remove_to(*name*)
Remove all elements from beginning to a specific element

This function is useful to create a sub-set of a struct.

param name Name of element to remove and all before element

type name str

serialize(*data*, *buffer=None*, *offset=0*)
Serialize dictionary into buffer

NOTE! If this is a union the method will try to serialize all the elements into the buffer (at the same position in the buffer). It is quite possible that the elements in the dictionary have contradicting data and the buffer

of the last serialized element will be ok while the others might be wrong. Thus you should only define the element that you want to serialize in the dictionary.

Parameters **data** (*dict*) – A dictionary keyed with element names. Elements can be omitted from the dictionary (defaults to value 0).

Returns A buffer that contains data

Return type bytearray

size()
Get size of structure or union.

Returns Number of bytes this structure represents alternatively largest of the elements (including end padding) if this is a union.

Return type int

6.2 BitfieldDef (bitfield representation)

class pycstruct.**BitfieldDef**(*byteorder='native'*, *size=-1*)
This class represents a bit field definition

The size of the bit field is 1, 2, 3, .., 8 bytes depending on the number of elements added to the bit field. You can also force the bitfield size by setting the size argument. When forcing the size larger bitfields than 8 bytes are allowed.

Parameters

- **byteorder** (*str, optional*) – Byte order of the bitfield. Valid values are 'native', 'little' and 'big'.
- **size** (*int, optional*) – Force bitfield to be a certain size. By default it will expand when new elements are added.

add(*name*, *nbr_of_bits=1*, *signed=False*)
Add a new element in the bitfield definition. The element will be added directly after the previous element.

The size of the bitfield will expand when required, but adding more than in total 64 bits (8 bytes) will generate an exception.

Parameters

- **name** (*str*) – Name of element. Needs to be unique.
- **nbr_of_bits** (*int, optional*) – Number of bits this element represents. Default is 1.
- **signed** (*bool, optional*) – Should the bit field be signed or not. Default is False.

assigned_bits()
Get size of bitfield in bits excluding padding bits

Returns Number of bits this bitfield represents excluding padding bits

Return type int

create_empty_data()
Create an empty dictionary with all keys

Returns A dictionary keyed with the element names. Values are "empty" or 0.

Return type dict

deserialize(*buffer*, *offset=0*)
Deserialize buffer into dictionary

Parameters

- **buffer** (*int*) – Buffer that contains the data to deserialize (1 - 8 bytes)
- **buffer_offset** – Start address in buffer

Returns A dictionary keyed with the element names

Return type dict

instance(*buffer=None*, *buffer_offset=0*)
Create an instance of this bitfield.

This is an alternative of using dictionaries and the *BitfieldDef.serialize()*/ *BitfieldDef.deserialize()* methods for representing the data.

Parameters

- **buffer** (*bytearray, optional*) – Byte buffer where data is stored. If no buffer is provided a new byte buffer will be created and the instance will be 'empty'.
- **buffer_offset** (*int, optional*) – Start offset in the buffer. This means that you can have multiple Instances (or other data) that shares the same buffer.

Returns A new Instance object

Return type *Instance()*

serialize(*data*, *buffer=None*, *offset=0*)
Serialize dictionary into buffer

Parameters **data** (*dict*) – A dictionary keyed with element names. Elements can be omitted from the dictionary (defaults to value 0).

Returns A buffer that contains data

Return type bytearray

size()
Get size of bitfield in bytes

Returns Number of bytes this bitfield represents

Return type int

6.3 EnumDef (enum representation)

class pycstruct.**EnumDef**(*byteorder='native'*, *size=-1*, *signed=False*)
This class represents an enum definition

The size of the enum is 1, 2, 3, .., 8 bytes depending on the value of the largest enum constant. You can also force the enum size by setting the size argument.

Parameters

- **byteorder** (*str, optional*) – Byte order of the enum. Valid values are 'native', 'little' and 'big'.
- **size** (*int, optional*) – Force enum to be a certain size. By default it will expand when new elements are added.

- **signed** (*bool, optional*) – True if enum is signed (may contain negative values)

add(*name*, *value=None*)
Add a new constant in the enum definition. Multiple constant might be assigned to the same value.

The size of the enum will expand when required, but adding a value requiring a size larger than 64 bits will generate an exception.

Parameters

- **name** (*str*) – Name of constant. Needs to be unique.
- **value** (*int, optional*) – Value of the constant. Automatically assigned to next available value (0, 1, 2, ...) if not provided.

deserialize(*buffer*, *offset=0*)
Deserialize buffer into a string (constant name)

If no constant name is defined for the value following name will be returned:

```
__VALUE__<value>
```

Where <value> is the integer stored in the buffer.

Parameters **buffer** (*bytearray*) – Buffer that contains the data to deserialize (1 - 8 bytes)

Returns The constant name (string)

Return type str

get_name(*value*)
Get the name representation of the value

Returns The constant name

Return type str

get_value(*name*)
Get the value representation of the name

Returns The value

Return type int

serialize(*data*, *buffer=None*, *offset=0*)
Serialize string (constant name) into buffer

Parameters **data** (*str*) – A string representing the constant name.

Returns A buffer that contains data

Return type bytearray

size()
Get size of enum in bytes

Returns Number of bytes this enum represents

Return type int

6.4 Parse source code files

pycstruct.**parse_file**(*input_files*, *byteorder='native'*, *castxml_cmd='castxml'*, *castxml_extra_args=None*, *cache_path=''*, *use_cached=False*, ***subprocess_kwargs*)

Parse one or more C source files (C or C++) and generate pycstruct instances as a result.

The result is a dictionary where the keys are the names of the struct, unions etc. typedef'ed names are also supported.

The values of the resulting dictionary are the actual pycstruct instance connected to the name.

This function requires that the external tool castxml is installed.

Alignment will automatically be detected and configured for the pycstruct instances.

Note that following pycstruct types will be used for char arrays:

- 'unsigned char []' = uint8 array
- 'signed char []' = int8 array
- 'char []' = utf-8 data (string)

Parameters

- **input_files** (*str or list*) – Source file name or a list of file names.
- **byteorder** (*str, optional*) – Byteorder of all elements Valid values are 'native', 'little' and 'big'. If not specified the 'native' byteorder is used.
- **castxml_cmd** (*str, optional*) – Path to the castxml binary. If not specified castxml must be within the PATH.
- **castxml_extra_args** (*list, optional*) – Extra arguments to provide to castxml. For example definitions etc. Check castxml documentation for which arguments that are supported.
- **cache_path** (*str, optional*) – Path where to store temporary files. If not provided, the default system temporary directory is used.
- **use_cached** (*boolean, optional*) – If this is True, use previously cached output from castxml to avoid re-running castxml (since it could be time consuming). Default is False.
- **subprocess_kwargs** – keyword arguments that will be passed down to the *subprocess.check_outputs()* call used to run castxml. By default, stderr will be redirected to stdout. To get the subprocess default behavior, pass *stderr=None*.

Returns A dictionary keyed on names of the structs, unions etc. The values are the actual pycstruct instances.

Return type dict

6.5 Parse source code strings

pycstruct.**parse_str**(*c_str*, *byteorder='native'*, *castxml_cmd='castxml'*, *castxml_extra_args=None*, *cache_path=''*, *use_cached=False*)

Parse a string containing C source code, such as struct or union defintions. Any valid C code is supported.

The result is a dictionary where the keys are the names of the struct, unions etc. typedef'ed names are also supported.

The values of the resulting dictionary are the actual pycstruct instance connected to the name.

This function requires that the external tool castxml is installed.

Alignment will automatically be detected and configured for the pycstruct instances.

Note that following pycstruct types will be used for char arrays:

- 'unsigned char []' = uint8 array
- 'signed char []' = int8 array
- 'char []' = utf-8 data (string)

Parameters

- **c_str** (*str*) – A string of C source code.
- **byteorder** (*str, optional*) – Byteorder of all elements Valid values are 'native', 'little' and 'big'. If not specified the 'native' byteorder is used.
- **castxml_cmd** (*str, optional*) – Path to the castxml binary. If not specified castxml must be within the PATH.
- **castxml_extra_args** (*list, optional*) – Extra arguments to provide to castxml. For example definitions etc. Check castxml documentation for which arguments that are supported.
- **cache_path** (*str, optional*) – Path where to store temporary files. If not provided, the default system temporary directory is used.
- **use_cached** (*boolean, optional*) – If this is True, use previously cached output from castxml to avoid re-running castxml (since it could be time consuming). Default is False.

Returns A dictionary keyed on names of the structs, unions etc. The values are the actual pycstruct instances.

Return type dict

6.6 Instance (instance of StructDef or BitfieldDef)

pycstruct.**Instance**(*datatype*, *buffer=None*, *buffer_offset=0*)

This class represents an Instance of either a *StructDef()* or a *BitfieldDef()*. The instance object contains a bytearray buffer where 'raw data' is stored.

The Instance object has following advantages over using dictionary objects:

- Data is only serialized/deserialized when accessed
- Data is validated for each element/attribute access. I.e. you will get an exception if you try to set an element/attribute to a value that is not supported by the definition.
- Data is accessed by attribute name instead of key indexing

Parameters

- **type** (*StructDef()* or *BitfieldDef()*) – The *StructDef()* class or *BitfieldDef()* class that we would like to create an instance of.

- **buffer** (*bytearray, optional*) – Byte buffer where data is stored. If no buffer is provided a new byte buffer will be created and the instance will be 'empty'.
- **buffer_offset** (*int, optional*) – Start offset in the buffer. This means that you can have multiple Instances (or other data) that shares the same buffer.

CHAPTER 7

Indices and tables

- genindex
- search

Index