ctyped Documentation

Release 0.8.0

Igor 'idle sign' Starikov

Nov 21, 2019

Contents

https://github.com/idlesign/ctyped

CHAPTER 1

Description

Build ctypes interfaces for shared libraries with type hinting

Requires Python 3.6+

- Less boilerplate;
- Logical structuring;
- Basic code generator (.so function -> ctyped function);
- Useful helpers.

CHAPTER 2

Requirements

1. Python 3.6+

CHAPTER 3

Table of Contents

3.1 Quickstart

```
from typing import Callable
from ctyped.toolbox import Library
from ctyped.types import CInt

# Define a library.
lib = Library('mylib.so')

# Structures are defined with the help of `structure` decorator
@lib.structure
class Box:

    one: int
    two: str
    innerbox: 'Box'  # That'll be a pointer.

# Type less with function names prefixes.
with lib.scope(prefix='mylib_'):

    # Describe function available in the library.
    @lib.function(name='otherfunc')
    def some_func(title: str, year: int) -> str:
        ...

    @lib.f  # `f` is a shortcut for function.
    def struct_func(src: Box) -> Box:
        ...

    with lib.s(prefix='mylib_grouped_', int_bits=64, int_sign=False):  # `s` is a␣
↪shortcut for scope.

        class Thing(CInt):
```

(continues on next page)

(continued from previous page)

```
            @lib.method(int_sign=True)  # Override `int_sign` from scope.
            def one(self, some: int) -> int:
                # Implicitly pass Thing instance alongside
                # with explicitly passed `some` arg.
                ...

            @lib.m  # `m` is a shortcut for method.
            def two(self, some:int, cfunc: Callable) -> int:
                # `cfunc` is a wrapper, calling an actual ctypes function.
                result = cfunc()
                # If no arguments, the wrapper will try to detect them automatically.
                return result + 1

    @lib.function
    def get_thing() -> Thing:
        ...

# Or you may use classes as namespaces.
@lib.cls(prefix='common_', str_type=CCharsW)
class Wide:

    @staticmethod
    @lib.function
    def get_utf(some: str) -> str:
        ...

# Bind ctype types to functions available in the library.
lib.bind_types()

# Call function from the library. Call ``mylib_otherfunc``
result_string = some_func('Hello!', 2019)
result_wide = Wide.get_utf('some')  # Call ``common_get_utf``

# Now to structures. Call ``mylib_struct_func``
mybox = struct_func(Box(one=35, two='dummy', innerbox=Box(one=100)))
# Let's pretend our function returns a box inside a box (similar to what's in the␣
↪params).
mybox.one  # Access box field value.
mybox.innerbox.one  # Access values from nested objects.

thing = get_thing()

thing.one(12)  # Call ``mylib_mylib_grouped_one``.
thing.two(13)  # Call ``mylib_mylib_grouped_two``
```

3.2 Sniffing

To save some time on function definition you can use `ctyped` automatic code generator.

It won't give you fully functional code, but is able to lower typing chore.

```
from ctyped.sniffer import NmSymbolSniffer
```

(continues on next page)

(continued from previous page)

```
# We sniff library first.
sniffer = NmSymbolSniffer('/here/is/my/libsome.so')
sniffed = sniffer.sniff()

# Now let's generate ctyped code.
dumped = sniffed.to_ctyped()

# At last we save autogenerated code into a file.
with open('library.py', 'w') as f:
    f.write(dumped)
```

There's also a shortcut to sniff an already defined library:

```
...
sniffed = lib.sniff()
dumped = result.to_ctyped()
```

3.3 Library

class ctyped.library.**Library**(*name: Union[str, pathlib.Path], *, autoload: bool = True, prefix: Optional[str] = None, str_type: Type[ctyped.types.CastedTypeBase] = <class 'ctyped.types.CChars'>, int_bits: Optional[int] = None, int_sign: Optional[bool] = None*)

Main entry point to describe C library interface.

Basic usage:

```
lib = Library('mylib')

with lib.scope(prefix='mylib_'):

    @lib.function()
    def my_func():
        ...

lib.bind_types()
```

Parameters

- **name** – Shared library name or filepath.
- **autoload** – Load library just on Library object initialization.
- **prefix** – Function name prefix to apply to functions in the library.

  Useful when C functions have common prefixes.
- **str_type** – Type to represent strings.
  - `CChars` - strings as chars (ANSI) **default**
  - `CCharsW` - strings as wide chars (UTF)

Note: This setting is global to library. Can be changed on function definition level.

- **int_bits** – int length to use by default.

  Possible values: 8, 16, 32, 64

  Note: This setting is global to library. Can be changed on function definition level.

- **int_sign** – Flag. Whether to use signed (True) or unsigned (False) ints.

  Note: This setting is global to library. Can be changed on function definition level.

bind_types()
Deduces ctypes argument and result types from Python type hints, binding those types to ctypes functions.

cls(*, *prefix: Optional[str] = None*, *str_type: Optional[ctyped.types.CastedTypeBase] = None*, *int_bits: Optional[int] = None*, *int_sign: Optional[bool] = None*)
Class decorator. Allows common parameters application for class methods.

```
@lib.cls(prefix='common_', str_type=CCharsW)
class Wide:

    @staticmethod
    @lib.function()
    def get_utf(some: str) -> str:
        ...
```

Parameters

- **prefix** – Function name prefix to apply to functions under the manager.
- **str_type** – Type to represent strings.
- **int_bits** – int length to be used in function.
- **int_sign** – Flag. Whether to use signed (True) or unsigned (False) ints.

f(*name_c: Union[str, Callable, None] = None*, *, *wrap: bool = False*, *str_type: Optional[ctyped.types.CastedTypeBase] = None*, *int_bits: Optional[int] = None*, *int_sign: Optional[bool] = None*) → Callable
Shortcut for `.function()`.

function(*name_c: Union[str, Callable, None] = None*, *, *wrap: bool = False*, *str_type: Optional[ctyped.types.CastedTypeBase] = None*, *int_bits: Optional[int] = None*, *int_sign: Optional[bool] = None*) → Callable
Decorator to mark functions which exported from the library.

Parameters

- **name_c** – C function name with or without prefix (see `.scope(prefix=)`). If not set, Python function name is used.
- **wrap** – Do not replace decorated function with ctypes function, but with wrapper, allowing pre- or post-process ctypes function call.

  Useful to organize functions to classes (to automatically pass `self`) to ctypes function to C function.

```
class Thing(CObject):

    @lib.function(wrap=True)
    def one(self, some: int) -> int:
        # Implicitly pass Thing instance alongside
        # with explicitly passed `some` arg.
        ...

    @lib.function(wrap=True)
    def two(self, some:int, cfunc: Callable) -> int:
        # `cfunc` is a wrapper, calling an actual ctypes function.
        # If no arguments provided the wrapper will try detect␣
↪them
        # automatically.
        result = cfunc()
        return result + 1
```

- **str_type** – Type to represent strings.

  Note: Overrides the same named param from library level (see __init__ description).

- **int_bits** – int length to be used in function.

  Note: Overrides the same named param from library level (see __init__ description).

- **int_sign** – Flag. Whether to use signed (True) or unsigned (False) ints.

  Note: Overrides the same named param from library level (see __init__ description).

load()
Loads shared library.

m(*name_c: Optional[str] = None*, ***kwargs*)
Shortcut for `.method()`.

method(*name_c: Optional[str] = None*, ***kwargs*)
Decorator. The same as `.function()` with `wrap=True`.

s = None
Shortcut for `.scope()`.

sniff() → ctyped.sniffer.SniffResult
Sniffs the library for symbols.

Sniffing result can be used as 'ctyped' code generator.

structure(***, *pack: Optional[int] = None*, *str_type: Optional[ctyped.types.CastedTypeBase] = None*, *int_bits: Optional[int] = None*, *int_sign: Optional[bool] = None*)
Class decorator for C structures definition.

```
@lib.structure
class MyStruct:

    first: int
```

(continues on next page)

(continued from previous page)

```
    second: str
    third: 'MyStruct'
```

Parameters

- **pack** – Allows custom maximum alignment for the fields (as #pragma pack(n)).
- **str_type** – Type to represent strings.
- **int_bits** – int length to be used in function.
- **int_sign** – Flag. Whether to use signed (True) or unsigned (False) ints.

3.4 Utils

class ctyped.utils.**ErrorInfo**(*num*, *code*, *msg*)
Create new instance of ErrorInfo(num, code, msg)

code
Alias for field number 1

msg
Alias for field number 2

num
Alias for field number 0

class ctyped.utils.**FuncInfo**(*name_py*, *name_c*, *annotations*, *options*)
Create new instance of FuncInfo(name_py, name_c, annotations, options)

annotations
Alias for field number 2

name_c
Alias for field number 1

name_py
Alias for field number 0

options
Alias for field number 3

ctyped.utils.**c_callback**(*use_errno: bool = False*) → Callable
Decorator to turn a Python function into a C callback function.

```
@lib.f
def c_func_using_callback(hook: CPointer) -> int:
    ...

@c_callback
def hook(num: int) -> int:
    return num + 10

c_func_using_callback(hook)
```

Parameters **use_errno** –

ctyped.utils.**get_last_error**() → ctyped.utils.ErrorInfo

Returns last error (errno) information named tuple:

```
(err_no, err_code, err_message)
```

Python Module Index

C

Index