PyCLibrary Documentation

Release 0.2.1

Luke Campagnola, Matthieu Dartiailh

Oct 10, 2022

CONTENTS

PyCLibrary tries to make wrapping dynamic librariries in python less cumbersome and more user friendly.

The idea is that most of the things needed such as constant values and function signatures are already presents in the headers files of the library (which are usually accessible as they are needed for using the library in C). So better to use them than copy everything by hand.

PyCLibrary includes 1) a pure-python C parser and 2) an automation library that uses C header file definitions to simplify the use of c bindings. The C parser currently processes all macros, typedefs, structs, unions, enums, function prototypes, and global variable declarations, and can evaluate typedefs down to their fundamental C types + pointers/arrays/function signatures. Pyclibrary can automatically build c structs/unions and perform type conversions when calling functions via cdll/windll.

PyCLibrary tries to present a ffi agnostic API to allow using different bindings. For the time being only the ctypes based backend is implemented but a cffi backend should be possible to implement (the rational for it would be that the CParser can be used on raw header files which are not always well supported by the cffi parser).

However if you need to manipulate the C object coming back from the library which cannot simply be mapped to Python object your code will most likely not be backend independent so it is discouraged to try to switch between backends.

PyCLibrary supports Python 2.7 and 3.3+

CHAPTER

ONE

GETTING STARTED

Getting started with PyCLibrary is easy. The next sections explain how to do so.

1.1 Installation

Contents

Installing PyCLibrary is straightforward. It is a pure python package and can be installed using pip (NB : for the time being there is no PyPI package so you must get a copy from github see *using_dev*):

```
$ pip install pyclibrary
```

It has a single mandatory dependency : pyparsing.

In order to run the testsuite you will also need py.test and if you want to build the docs you will need sphinx (>1.3 current development version). All those can be installed through pip using the following commands:

```
$ pip install py.test
$ pip install sphinx
```

1.1.1 Testing your installation

To test your installation open a python interpreter and import pyclibrary.

```
>>> import pyclibrary
```

The last command will have no output if everything went well.

If you encounter any problem, take a look at the *FAQS*. If everything fails, feel free to open an issue in our issue_tracker.

1.1.2 Using the development version

You can install the development version directly from Github:

```
$ pip install https://github.com/MatthieuDartiailh/pyclibrary/zipball/master
```

1.2 Basic usage

Contents

This guide tries to give a simple overview of PyCLibrary capabilities. At the end it refers to C concepts as some basic knowledge of them might be necessary when interfacing a C library but do not be scared by them.

1.2.1 Parsing headers

The first step you should take when trying to interface with a dynamic library using PyCLibrary is to check that it can correctly Parse the header files:

```
>>> from pyclibrary import CParser
>>> parser = CParser(['first_header_file_path','second_header_file_path'])
>>> print(parser)
```

If the second command does not raise any issue it means that it successfully parsed the headers. However even in such a case the parser might have overlooked some definitions. The last command will print all the definitions extracted from the header files grouped by categories :

- types : the custom types defined in the headers
- variables : the global variable of the libraries.
- fnmacros : the function macros declared in the headers (Those are used by the compiler preprocessor and you have no reason to access them).
- macros : the macros defined in the headers.
- structs : the custom structures used by the libraries.
- unions : the customs unions used by the libraries.
- enums : the enumerations defined in the headers.
- functions : the functions you will be able to call.

- values : the global values you may need to access (mainly macro values which are used to provide a more descriptive representation of integer values)

You can quickly go over them and check if something is missing.

Note: On windows, it is generally a good idea to include some standards windows definition. To do so pass the result of calling the win_defs function to the parser as second argument (copy_from keyword).

Note: The *CParser* does not handle the include directive (for the time being) so you must pass all the header files.

Caching the parsed files

As parsing the headers is a fairly expensive process, it is a good idea to cache the parsed definition.

To cache definitions, you simply have to provide a path pointing to the file in which to save the parsed definitions to the parser (cache keyword). If the cache file already exists, it is loaded only if the version of the parser matches (which allows update to always take effects) and if the arguments passed to the *CParser* are the same (if you ask for different replacements in your file it will trigger a re-parsing).

The previous procedure should be sufficient in general but in some cases you might want a finer control on the parsing procedure. See for a more detailed explanation.

1.2.2 Binding the library

Once you know that you can correctly parse the headers of your library, you are ready to bind it. To do so, you must create a *CLibrary* object:

```
    >>> from pyclibrary import CLibrary
    >>> clib = CLibrary('mylibrary.dll', parser, prefix='Lib_',
>>>                 lock_calls=False, convention='cdll', backend='ctypes')
```

In order to work, the *CLibrary* needs the name or the path to the library to use (a .dll on Windows, a .so on Linux), and either an initialized parser or a list of header files which will be parsed for definitions. When you provide simply name of the library it is looked for in standard locations according to your OS. All other keyword arguments are optionals :

- **prefix :**
  prefix or list of prefix often found in the library function or attributes names. This allow you to access to them without the prefix, while not preventing to use the complete name.
- **lock_calls :**
  when this flag is set all calls to the dll are made thread safe by acquiring a lock before calling and releasing it after. This can be useful if the library is not thread-safe.
- **convention :**
  this only applies on windows platform where the calling convention can be either 'cdll' (Linux standard), windll or oledll. Note that all conventions might not be supported on all platforms and with all backends.
- **backend :**
  the name of the backend to use when binding to the library. Currently the only backend relies on the ctypes library, in the future one using the cffi library might be used.

All other keyword arguments will be passed to when creating a *CParser* if a list of headers files is passed.

You now have access to all the attributes, types and functions defined by the library.

Accessing attributes

The preferred way to access library attributes is simply by using the . syntax:

```
>>> clib.HIGH_FLAG
1
```

This simply looked for into all the known definition for a HIGH_FLAG value or Lib_HIGH_FLAG value as we specified 'Lib_' as a prefix. This will work for values, functions, types, structures, unions, enumerations but not for macros definitions.

But you can also specify what kind of object you are looking for using the following syntax:

```
>>> clib('values', 'HIGH_FLAG')
1
```

The recognized values for the first argument are the following : 'values', 'functions', 'types', 'structs', 'unions', or 'enums'. This method is roughly equivalent to the first one. It is however useful if for example one needs to access to an enumeration type : when looking for it the entries found in values which specifies the mapping between names and their integer value is always found first (as it is most of the time what is useful), so if you want the type you need to specify it explicitly.

The third way gives access directly to the parser definitions:

```
>>>clib['values']['HIGH_FLAG']
1
```

This is equivalent to doing:

```
>>>parser.defs['values']['HIGH_FLAG']
```

Calling functions

One usual behavior of C function is to return a kind of flag signaling that the operation while returning the real values of interest by updating pointers which have been passed to them. Most of the time those pointer does not need to be initialized to any particular value and it is often tedious to create them. PyCLibrary tries to make that kind of things easier. Here are some of the key concept used :

- function always return a *CallResult* object which encapsulates the return value of the function and all the arguments passed to it.
- when calling a function you can use keyword arguments based on the C signature of the function.
- you can omit all uninitialized pointers the function expects, PyCLibrary will create them for you and they will be accessible in the *CallResult* object.

Let's consider a C function whose signature is the following :

```
RETURN_CODE get_library_version(U8 *Major,U8 *Minor,U8 *Revision);
```

Once wrapped by PyCLibrary this function can be called as follows:

```
>>> ret = clib.get_library_version()
>>> ret()
1  # This is the RETURN_CODE value, 1 means the call succeeded
>>> ret[0]
```

(continues on next page)

(continued from previous page)

```
0  # This is the major version.
>>> ret['Minor']
1
```

Some explanations :

- first we call the function, not providing any pointers and store the CallResult object.
- then we query the return value by calling the *CallResult* object. When doing this PyCLibrary tries to convert the value to a nice Python equivalent and if it is not possible it returns the underlying backend object.
- finally we access to the major and minor version info. To access to the major version info we query the argument using its index, for the minor we use the name of the argument.

Sometimes even if a Python equivalent exists you might need to access to raw backend objects. You can find it in the attribute *CallResult.rval* for the return value and in *CallResult.rval* for the argument (that you passed and the created pointers).

Note that all the pointers automatically created by PyCLibrary are dereferenced automatically so that you get the value to which they point to, when accessed through the '[]' operation, or using tuple unpacking see below.

As this syntax is not always convenient when we need to proceed to many calls the *CallResult* object can be used as an iterator to allow unpacking:

```
>>> res, (major, minor, rev) = clib.get_library_version()
>>> '{}.{}.{}'.format(major, minor, rev)
'0.1.0'
```

Note that the arguments are unpacked as a tuple (actually a generator) which makes it easy to ignore it if the function directly return the value you want:

```
>>> val, _ = clib.get_value()
2
```

Note: The value auto-generated are pointers but are not returned as such because most of the time it is the stored value that is needed. For pointers of pointers which generally represents arrays, it dereference only the external pointer so that the array element can be accessed using pointer[i] (which is a valid C syntax). This magic happens only with auto-generated values, if you manually pass a pointer the value in the arguments will be a pointer.

Creating and passing arrays

One special case of passing values by reference (ie using a pointer) is the case of the arrays. Here two solutions exist depending on the behavior of the library :

- the function expects a pointer to pointer and handles itself the memory allocation.
- the function expects a pointer to an already existing array, and will use it or modify it.

In the first case, you can let PyCLibrary handle everything, you will get a pointer that you can index like any iterable (but you can't determine its length, you must get that information from the library in another way). In the second case you cannot just let PyCLibrary creates the pointer because when the function will write in the array it might access memory it should not and corrupt data because the memory was never allocated. For this case, PyCLibrary provides the *build_array* helper function. This function takes as arguments the library object, the type of the data to store in the array (as a str or as type object) and the shape of the array to build (multidimensional arrays are supported), and optionally an initialization iterable (for one dimensional arrays only).

Let's consider two functions:

```
void fill_array(int *array);
void allocate_array(int size, int **array);
```

Note that without reading the docs, you cannot know that fill_array needs an array and not simply a pointer to an integer. You must read the docs !

And here it the interfacing code:

```
>>> arr = build_array(clib, 'int', 5)
>>> _, (arr) = fill_array(arr)
>>> [arr[i] for i in range(5)]
[0, 10, 20, 21, 55]
>>> _, (size, arr) = allocate_array()
>>> [arr[i] for i in range(size)]
[-1, 2, 5, 8, -9]
```

This is fairly straightforward, simply note that you can directly pass the array in place of a pointer, the backend handle the conversion.

1.3 Configuration

Contents

Most of the time the default configuration of PyCLibrary should be sufficient. However it may not always be so. Here are some ways to tweak it to your needs.

1.3.1 Specifying headers and libraries locations

When parsing numerous headers or using PyCLibrary in an application, it might prove tedious to always specifies the full path to headers files (the same can apply to libraries if their not located in a standard location). PyCLibrary allows you to add folder in which to look into for headers and libraries.

Consider a case in which you store the headers in a 'headers' folder by your script, and the library into a 'lib' folder :

```
import os
from pyclibrary import add_header_locations, add_library_locations

curr_dir = os.path.dirname(__file__)
add_header_locations([os.path.join(curr_dir, 'headers')])
add_library_locations([os.path.join(curr_dir, 'lib')])

parser = CParser('my_lib_header.h')
clib = CLibrary('my_lib.so', parser)
```

Note: PyCLibrary does not explore sub-folders when looking for headers and library (it might in the future).

1.3.2 Manual initialization

The first time you create a CParser or CLibrary, PyCLibrary does some initialization based on your OS. It basically defines the standard known ctypes and the specific modifiers supported by the compiler (things like __stdcall for example).

You can manually initialize the CParser and the CLibrary by calling the initialization function found in pyclibrary and specifies your own types and modifiers. You can also use the auto_init function to add things on top of your OS specific stuff.

CHAPTER

TWO

ARCHITECTURE REFERENCES

UNDER CONSTRUCTION

CHAPTER

THREE

FAQS

3.1 Contributing to PyCLibrary

You can contribute in different ways:

3.1.1 Report issues

You can report any issues with the package, the documentation to the PyCLibrary issue tracker. Also feel free to submit feature requests, comments or questions.

3.1.2 Contribute code

To contribute fixes, code or documentation to PyCLibrary, fork PyCLibrary in github and submit the changes using a pull request.

In any case, feel free to use the issue tracker to discuss ideas for new features or improvements.

CHAPTER

FOUR

API REFERENCES

4.1 Subpackages

4.1.1 pyclibrary.backends package

Submodules

pyclibrary.backends.ctypes module

Proxy to both CHeader and ctypes, allowing automatic type conversion and function calling based on C header definitions.

pyclibrary.backends.ctypes.**make_mess**(*mess*)

class pyclibrary.backends.ctypes.**CTypesCLibrary**(*lib*, **args*, ***kwargs*)

Bases: *CLibrary*

The CLibrary class is intended to automate much of the work in using ctypes by integrating header file definitions from CParser.

This class serves as a proxy to a ctypes object, adding a few features:

- allows easy access to values defined via CParser
- automatic type conversions for function calls using CParser function signatures
- creates ctype classes based on type definitions from CParser

Initialize using a ctypes shared object and a CParser:
headers = CParser.winDefs() lib = CLibrary(windll.User32, headers)

There are 3 ways to access library elements:

lib(type, name):
type can be one of 'values', 'functions', 'types', 'structs', 'unions', or 'enums'. Returns an object matching name. For values, the value from the headers is returned. For functions, a callable object is returned that handles automatic type conversion for arguments and return values. For structs, types, and enums, a ctypes class is returned matching the type specified.

lib.name:
searches in order through values, functions, types, structs, unions, and enums from header definitions and returns an object for the first match found. The object returned is the same as returned by lib(type, name). This is the preferred way to access elements from CLibrary, but may not work in some situations (for example, if a struct and variable share the same name).

lib[type]:
Accesses the header definitions directly, returns definition dictionaries based on the type requested. This is equivalent to headers.defs[type].

Parameters

- **lib** – Library object.
- **headers** (CParser) – CParser holding all the definitions.
- **prefix** (*unicode*) – Prefix to remove from all definitions.
- **fix_case** (*bool*) – Should name be converted from camelCase to python PEP8 compliants names.

Null = <object object>
Balise to use when a NULL pointer is needed

backend = 'ctypes'
Id of the backend

pyclibrary.backends.ctypes.**init_clibrary**(*extra_types={}*)

pyclibrary.backends.ctypes.**identify_library**(*lib*)

pyclibrary.backends.ctypes.**get_library_path**(*lib*)

4.2 Submodules

4.2.1 pyclibrary.c_library module

Proxy to library object, allowing automatic type conversion and function calling based on C header definitions.

pyclibrary.c_library.**make_mess**(*mess*)

class pyclibrary.c_library.**CLibraryMeta**(*name*, *bases*, *dct*)
Bases: type

Meta class responsible for determining the backend and ensuring no duplicates libraries exists.

backends = {'ctypes': <class 'pyclibrary.backends.ctypes.CTypesCLibrary'>}

libs = <WeakValueDictionary>

class pyclibrary.c_library.**CLibrary**(*lib*, **args*, ***kwargs*)
Bases: object

The CLibrary class is intended to automate much of the work in using ctypes by integrating header file definitions from CParser. This class serves as a proxy to a backend, adding a few features:

- allows easy access to values defined via CParser.
- automatic type conversions for function calls using CParser function signatures.
- creates ctype classes based on type definitions from CParser.

Initialize using a ctypes shared object and a CParser: >>> headers = CParser.winDefs() >>> lib = CLibrary(windll.User32, headers)

There are 3 ways to access library elements:

- **lib(type, name):**
  type can be one of 'values', 'functions', 'types', 'structs', 'unions', or 'enums'. Returns an object matching name. For values, the value from the headers is returned. For functions, a callable object is returned that handles automatic type conversion for arguments and return values. For structs, types, and enums, a ctypes class is returned matching the type specified.
- **lib.name:**
  searches in order through values, functions, types, structs, unions, and enums from header definitions and returns an object for the first match found. The object returned is the same as returned by lib(type, name). This is the preferred way to access elements from CLibrary, but may not work in some situations (for example, if a struct and variable share the same name).
- **lib[type]:**
  Accesses the header definitions directly, returns definition dictionaries based on the type requested. This is equivalent to headers.defs[type].

Parameters

- **lib** – Library object.
- **headers** (*unicode or CParser*) – Path to the header files or CParser holding all the definitions.
- **prefix** (*unicode, optional*) – Prefix to remove from all definitions.
- **lock_calls** (*bool, optional*) – Whether or not to lock the calls to the underlying library. This should be used only if the underlying library is not thread safe.
- **convention** (*{'cdll', 'windll', 'oledll'}*) – Calling convention to use. Not all backends supports all calling conventions.
- **backend** (*unicode, optional*) – Name of the backend to use. This is ignored if an already initialised library object is passed. NB : this kwarg is used by the metaclass.
- **kwargs** – Additional keywords argument which are passed to the CParser if one is created.

Null = <object object>

Balise to use when a NULL pointer is needed

class pyclibrary.c_library.**CFunction**(*lib*, *func*, *sig*, *name*, *lock_call*)

Bases: object

Wrapper object for a function from the library.

arg_c_type(*arg*)

Return the type required for the specified argument.

Parameters
arg (*int or unicode*) – Name or index of the argument whose type should be returned.

pretty_signature()

class pyclibrary.c_library.**CallResult**(*lib*, *rval*, *args*, *sig*, *guessed*)

Bases: object

Class for bundling results from C function calls.

Allows access to the function value as well as all of the arguments, since the function call will often return extra values via these arguments:

- Original ctype objects can be accessed via result.rval or result.args

- Python values carried by these objects can be accessed using ()

To access values:

- The return value: ()
- The nth argument passed: [n]
- The argument by name: ['name']
- All values that were auto-generated: .auto()

The class can also be used as an iterator, so that tuple unpacking is possible:

```
>>> ret, (arg1, arg2) = lib.run_some_function(...)
```

lib

Reference to the CLibrary to which the function that created this object balongs.

Type
CLibrary

rval

Value returned by the C function.

args

Arguments passed to the C function.

Type
tuple

sig

Signature of the function which created this object.

guessed

Pointers that were created on the fly.

Type
tuple

find_arg(*arg*)

Find argument based on name.

auto()

Return a list of all the auto-generated values.

Pointers are dereferenced.

pyclibrary.c_library.**cast_to**(*lib*, *obj*, *typ*)

Cast obj to a new type.

Parameters

- **lib** (CLibrary) – Reference to the library to which the object 'belongs'. This is needed as the way to get the address depends on the backend.
- **obj** – Object whose address should be returned.

type
[type or string] Type object or string which will be used to determine the type of the array elements.

pyclibrary.c_library.**build_array**(*lib*, *typ*, *size*, *vals=None*)

Build an array of the specified type and the specified size.

Parameters

- **lib** (*CLibrary*) – Reference to the library with which this object will be used. This is needed as the way to build the array depends on the backend.
- **type** (*type or string*) – Type object or string which will be used to determine the type of the array elements.
- **size** (*int or tuple*) – Dimensions of the array to create.
- **vals** (*list, optional*) – Initial values with which to fill the array.

4.2.2 pyclibrary.c_parser module

Used for extracting data such as macro definitions, variables, typedefs, and function signatures from C header files.

pyclibrary.c_parser.**win_defs**(*version='1500'*)

Loads selection of windows headers included with PyCLibrary.

These definitions can either be accessed directly or included before parsing another file like this: >>> windefs = c_parser.win_defs() >>> p = c_parser.CParser("headerFile.h", copy_from=windefs)

Definitions are pulled from a selection of header files included in Visual Studio (possibly not legal to distribute? Who knows.), some of which have been abridged because they take so long to parse.

Parameters
version (*unicode*) – Version of the MSVC to consider when parsing.

Returns
parser – CParser containing all the infos from te windows headers.

Return type
CParser

class pyclibrary.c_parser.**CParser**(*files=None*, *copy_from=None*, *replace=None*, *process_all=True*, *cache=None*, *check_cache_validity=True*, ***kwargs*)

Bases: object

Class for parsing C code to extract variable, struct, enum, and function declarations as well as preprocessor macros.

This is not a complete C parser; instead, it is meant to simplify the process of extracting definitions from header files in the absence of a complete build system. Many files will require some amount of manual intervention to parse properly (see 'replace' and extra arguments)

Parameters

- **files** (*str or iterable, optional*) – File or files which should be parsed.
- **copy_from** (*CParser or iterable of CParser, optional*) – CParser whose definitions should be included.
- **replace** (*dict, optional*) – Specify som string replacements to perform before parsing. Format is {'searchStr': 'replaceStr', ...}
- **process_all** (*bool, optional*) – Flag indicating whether files should be parsed immediatly. True by default.

- **cache** (*unicode, optional*) – Path of the cache file from which to load definitions/to which save definitions as parsing is an expensive operation.
- **check_cache_validity** (*bool, optional*) – Flag indicating whether to perform validity checking when using a cache file. This is useful in a scenario where the python wrapper needs to be used without access to the headers
- **kwargs** – Extra parameters may be used to specify the starting state of the parser. For example, one could provide a set of missing type declarations by types={'UINT': ('unsigned int'), 'STRING': ('char', 1)} Similarly, preprocessor macros can be specified: macros={'WINAPI': ''}

Example

Create parser object, load two files

```
>>> p = CParser(['header1.h', 'header2.h'])
```

Remove comments, preprocess, and search for declarations

```
>>> p.process_ all()
```

Just to see what was successfully parsed from the files

```
>>> p.print_all()
```

Access parsed declarations

```
>>> all_values = p.defs['values']
>>> functionSignatures = p.defs['functions']
```

To see what was not successfully parsed

```
>>> unp = p.process_all(return_unparsed=True)
>>> for s in unp:
        print s
```

cache_version = 2

process_all(*cache=None*, *return_unparsed=False*, *print_after_preprocess=False*, *check_cache_validity=True*)

Remove comments, preprocess, and parse declarations from all files.

This operates in memory, and thus does not alter the original files.

Parameters

- **cache** (*unicode, optional*) – File path where cached results are be stored or retrieved. The cache is automatically invalidated if any of the arguments to __init__ are changed, or if the C files are newer than the cache.
- **return_unparsed** (*bool, optional*) – Passed directly to parse_defs.
- **print_after_preprocess** (*bool, optional*) – If true prints the result of preprocessing each file.

Returns

results – List of the results from parse_defs.

Return type
list

`load_cache`(*cache_file*, *check_validity=False*)

Load a cache file.

Used internally if cache is specified in process_all().

Parameters

- **cache_file** (*unicode*) – Path of the file from which the cache should be loaded.
- **check_validity** (*bool, optional*) –

  If True, then run several checks before loading the cache:

  - cache file must not be older than any source files
  - cache file must not be older than this library file
  - options recorded in cache must match options used to initialize CParser

Returns
result – Did the loading succeeded.

Return type
bool

`import_dict`(*data*)

Import definitions from a dictionary.

The dict format should be the same as CParser.file_defs. Used internally; does not need to be called manually.

`write_cache`(*cache_file*)

Store all parsed declarations to cache. Used internally.

`find_headers`(*headers*)

Try to find the specified headers.

`load_file`(*path*, *replace=None*)

Read a file, make replacements if requested.

Called by __init__, should not be called manually.

Parameters

- **path** (*unicode*) – Path of the file to load.
- **replace** (*dict, optional*) – Dictionary containing strings to replace by the associated value when loading the file.

`print_all`(*filename=None*)

Print everything parsed from files. Useful for debugging.

Parameters
filename (*unicode, optional*) – Name of the file whose definition should be printed.

`remove_comments`(*path*)

Remove all comments from file.

Operates in memory, does not alter the original files.

preprocess(*path*)

Scan named file for preprocessor directives, removing them while expanding macros.

Operates in memory, does not alter the original files.

Currently support : - conditionals : ifdef, ifndef, if, elif, else (defined can be used in a if statement). - definition : define, undef - pragmas : pragma

eval_preprocessor_expr(*expr*)

process_macro_defn(*t*)

Parse a #define macro and register the definition.

compile_fn_macro(*text*, *args*)

Turn a function macro spec into a compiled description.

expand_macros(*line*)

Expand all the macro expressions in a string.

Faulty calls to macro function are left untouched.

expand_fn_macro(*name*, *text*)

Replace a function macro.

parse_defs(*path*, *return_unparsed=False*)

Scan through the named file for variable, struct, enum, and function declarations.

Parameters

- **path** (*unicode*) – Path of the file to parse for definitions.
- **return_unparsed** (*bool, optional*) – If true, return a string of all lines that failed to match (for debugging purposes).

Returns

tokens – Entire tree of successfully parsed tokens.

Return type

list

build_parser()

Builds the entire tree of parser elements for the C language (the bits we support, anyway).

process_declarator(*decl*)

Process a declarator (without base type) and return a tuple (name, [modifiers])

See process_type(...) for more information.

process_type(*typ*, *decl*)

Take a declarator + base type and return a serialized name/type description.

The description will be a list of elements (name, [basetype, modifier, modifier, ...]):

- name is the string name of the declarator or None for an abstract declarator
- basetype is the string representing the base type
- modifiers can be:
  - * : pointer (multiple pointers *** allowed)
  - *&* : reference
  - *__X* : calling convention (windows only). X can be *cdecl* or *stdcall*

- list : array. Value(s) indicate the length of each array, -1 for incomplete type.
- tuple : function, items are the output of processType for each function argument.

Examples

- int *x[10]* => *('x', ['int', [10], ''])*
- char fn(int x) => ('fn', ['char', [('x', ['int'])]])
- struct s (*)(int, int*) => (None, ["struct s", ((None, ['int']), (None, ['int', '*'])), '*'])

process_enum(*s*, *l*, *t*)

process_function(*s*, *l*, *t*)

Build a function definition from the parsing tokens.

packing_at(*line*)

Return the structure packing value at the given line number.

process_struct(*s*, *l*, *t*)

process_variable(*s*, *l*, *t*)

process_typedef(*s*, *l*, *t*)

eval_expr(*toks*)

Evaluates expressions.

Currently only works for expressions that also happen to be valid python expressions.

eval(*expr*, **args*)

Just eval with a little extra robustness.

add_def(*typ*, *name*, *val*)

Add a definition of a specific type to both the definition set for the current file and the global definition set.

rem_def(*typ*, *name*)

Remove a definition of a specific type to both the definition set for the current file and the global definition set.

is_fund_type(*typ*)

Return True if this type is a fundamental C type, struct, or union.

ATTENTION: This function is legacy and should be replaced by Type.is_fund_type()

eval_type(*typ*)

Evaluate a named type into its fundamental type.

ATTENTION: This function is legacy and should be replaced by Type.eval()

find(*name*)

Search all definitions for the given name.

find_text(*text*)

Search all file strings for text, return matching lines.

4.2.3 pyclibrary.errors module

Errors that can happen during parsing or binding.

exception pyclibrary.errors.**PyCLibError**

Bases: Exception

Base exception for all PyCLibrary exceptions.

exception pyclibrary.errors.**DefinitionError**

Bases: *PyCLibError*

Excepion signaling that one definition found in the header is malformed or meaningless.

4.2.4 pyclibrary.init module

Initialisation routines.

Those should be run before creating a CParser and can be run only once. They are used to declare additional types and modifiers for the parser.

pyclibrary.init.**init**(*extra_types=None*, *extra_modifiers=None*)

Init CParser and CLibrary classes.

Parameters

- **extra_types** (*dict, optional*) – typeName->c_type pairs to extend typespace.
- **extra_modifiers** (*list, optional*) – List of modifiers, such as '__stdcall'.

pyclibrary.init.**auto_init**(*extra_types=None*, *extra_modifiers=None*, *os=None*)

Init CParser and CLibrary classes based on the targeted OS.

Parameters

- **extra_types** (*dict, optional*) – Extra typeName->c_type pairs to extend typespace.
- **extra_modifiers** (*list, optional*) – List of extra modifiers, such as '__stdcall'.
- **os** (*{'win32', 'linux2', 'darwin'}, optional*) – OS for which to prepare the system. If not specified sys is used to identify the OS.

4.2.5 pyclibrary.utils module

Utility functions to retrieve headers or library path and architecture.

Most of those function have been taken or adapted from the ones found in PyVISA.

Functions

find_header : Find the path to a header file. find_library : Find the path to a shared library from its name.

pyclibrary.utils.**add_header_locations**(*dir_list*)

Add directories in which to look for header files.

pyclibrary.utils.**find_header**(*h_name*, *dirs=None*)

Look for a header file.

Headers are looked for in the directories specified by the user using the add_header_locations function, in the headers directory of PyCLibrary, and in the standards locations according to the operation system.

Parameters

- **h_name** (*unicode*) – Name of the header to retrieve (should include the ".h")
- **dirs** (*list, optional*) – List of directory which should be searched for the header in addition to the default ones.

Returns
path – Path to the header file.

Return type
unicode

:raises OSError : if no matching file can be found.:

pyclibrary.utils.**add_library_locations**(*dir_list*)

Add directories in which to look for libraries.

pyclibrary.utils.**find_library**(*name*, *dirs=None*)

Look for a library file.

Libraries are looked for in the directories specified by the user using the add_library_locations function, and using the find_library function found the thirdparty package.

Parameters

- **name** (*unicode*) – Name of the library to retrieve (should include the extension)
- **dirs** (*list, optional*) – List of directory which should be searched for the library before ressorting to using thirdparty.find_library.

Returns
path – Path to the library file.

Return type
unicode

:raises OSError : if no matching file can be found.:

class pyclibrary.utils.**LibraryPath**(*path*, *found_by='auto'*)

Bases: str

property arch

property is_32bit

property is_64bit

property bitness

pyclibrary.utils.**get_arch**(*filename*)

pyclibrary.utils.**get_shared_library_arch**(*filename*)

pyclibrary.utils.**check_output**(**popenargs*, ***kwargs*)

Run command with arguments and return its output as a byte string.

Backported from Python 2.7 as it's implemented as pure python on stdlib.

```
>>> check_output(['/usr/bin/python', '--version'])
Python 2.6.2
```

- *Getting started*

  How to set up PyCLibrary and make your first step with it.

- *Architecture References*

  More references on PyCLibrary internals.

- *FAQS*

  Some questions that might have occurred to others too.

- *API references*

  When all else fails, consult the API docs to find the answer you need. The API docs also include convenient links to the most definitive PyCLibrary documentation: the source.

PYTHON MODULE INDEX

p

INDEX

L

M

N

P

R

S

W