

---

# **binary\_c-python**

**Jeff Andrews, Robert Izzard, David Hendriks**

**Nov 30, 2019**



## CONTENTS:

<b>1</b>	<b>Python module for binary_c</b>	<b>1</b>
1.1	Requirements . . . . .	1
1.2	Environment variables . . . . .	1
1.3	Build instructions . . . . .	1
1.4	Usage notes . . . . .	2
<b>2</b>	<b>examples</b>	<b>3</b>
<b>3</b>	<b>binaryc_python_utils</b>	<b>5</b>
3.1	custom_logging_functions module . . . . .	5
3.2	functions module . . . . .	6
<b>4</b>	<b>Indices and tables</b>	<b>7</b>
	<b>Python Module Index</b>	<b>9</b>
	<b>Index</b>	<b>11</b>



## PYTHON MODULE FOR BINARY\_C

Based on a original work by Jeff Andrews (can be found in `old_solution/` directory) updated and extended for Python3 by Robert Izzard, David hendriks

Warning : THIS CODE IS EXPERIMENTAL!

r.izzard@surrey.ac.uk [http://personal.ph.surrey.ac.uk/~ri0005/binary\\_c.html](http://personal.ph.surrey.ac.uk/~ri0005/binary_c.html) 09/06/2019

### 1.1 Requirements

- Python3
- `binary_c` version 2.1+
- `requirements.txt` (no?)

### 1.2 Environment variables

Before compilation you should set the following environment variables:

- required: `BINARY_C` should point to the root directory of your `binary_c` installation
- recommended: `LD_LIBRARY_PATH` should include `$BINARY_C/src` and whatever directories are required to run `binary_c` (e.g. locations of `libgsl`, `libmemoize`, `librinterpolate`, etc.)
- recommended: `LIBRARY_PATH` should include whatever directories are required to build `binary_c` (e.g. locations of `libgsl`, `libmemoize`, `librinterpolate`, etc.)

### 1.3 Build instructions

To build the module, make sure you have built `binary_c` (with `make` in the `binary_c` root directory), its shared library (with `make libbinary_c.so` in the `binary_c` root directory), and set environment variables as described above, then run the following code in t:

```
make clean
make
```

Then to test the Python module:

```
python3 ./python_API_test.py
```

You will require whatever libraries with which `binary_c` was compiled, as well as the compiler with which Python was built (usually `gcc`, which is easily installed on most systems).

If you want to be able to import the `binary_c` module correctly for child directories (or anywhere for that matter), execute or put the following code in your `.bashrc/.zshrc`:

```
export LD_LIBRARY_PATH=<full path to directory containing libbinary_c_api.so>:$LD_
↪LIBRARY_PATH
export PYTHONPATH=<full path to directory containing libbinary_c_api.so>:$PYTHONPATH
```

## 1.4 Usage notes

When running a jupyter notebook and importing `binary_c`, it might happen that the module `binary_c` cannot be found. I experienced this when I executed Jupyter Notebook from a virtual environment which didnt use the same python (version/binary/shim) as the one I built this library with. Make sure jupyter does use the same underlying python version/binary/shim. That resolved the issue for me.

Also: I figured that having `binaryc` output the log like “`t=10e4 ...`” (i.e. printing the parameter names as well as their values) would be useful because in that way one can easily have python read that out automatically instead of having to manually copy the list of parameter names.

See `examples/ dir` for some working examples

## EXAMPLES

This chapter serves to document several of the example usages

`examples.examples.run_example_binary()`

Function to run a binary system. Very basic approach which directly addresses the `run_binary(..)` python-c wrapper function.

`examples.examples.run_example_binary_with_custom_logging()`

Function that will use a automatically generated piece of logging code. Compile it, load it into memory and run a binary system. See `run_system` on how several things are done in the background here.

`examples.examples.run_example_binary_with_run_system()`

This function serves as an example on the function `run_system` and `parse_output`. There is more functionality with this method and several tasks are done behind the scene.

Requires pandas, numpy to run.

`run_system`: mostly just makes passing arguments to the function easier. It also loads all the necessary defaults in the background  
`parse_output`: Takes the raw output of `binary_c` and selects those lines that start with the given header. Note, if you dont use the `custom_logging` functionality `binary_c` should be configured to have output that starts with that given header

The parsing of the output only works correctly if either all of the values are described inline like `'mass=<number>'` or none of them are.

`examples.examples.run_example_binary_with_writing_logfile()`

Same as above but when giving the `log_filename` argument the log filename will be written



## BINARYC\_PYTHON\_UTILS

### 3.1 custom\_logging\_functions module

`custom_logging_functions.autogen_C_logging_code` (*logging\_dict*)

Function that autogenerates PRINTF statements for binaryc. input is a dictionary where the key is the header of that logging line and items which are lists of parameters that will be put in that logging line

Example:

```
{ 'MY_STELLAR_DATA':  
  [  
    'model.time',  
    'star[0].mass',  
    'model.probability',  
    'model.dt'  
  ]  
}
```

`custom_logging_functions.binary_c_log_code` (*code*)

Function to construct the code to construct the custom logging function

`custom_logging_functions.binary_c_write_log_code` (*code, filename*)

Function to write the generated logging code to a file

`custom_logging_functions.compile_shared_lib` (*code, sourcefile\_name, outfile\_name, verbose=False*)

Function to write the custom logging code to a file and then compile it.

`custom_logging_functions.create_and_load_logging_function` (*custom\_logging\_code*)

Function to automatically compile the shared library with the given custom logging code and load it with ctypes

**returns:** memory address of the custom logging function in a int type.

`custom_logging_functions.from_binary_c_config` (*config\_file, flag*)

Function to run the binaryc\_config command with flags

`custom_logging_functions.return_compilation_dict` (*verbose=False*)

Function to build the compile command for the shared library

inspired by `binary_c_inline_config` command in perl

TODO: this function still has some cleaning up to do wrt default values for the compile command # <https://developers.redhat.com/blog/2018/03/21/compiler-and-linker-flags-gcc/>

**returns:**

- string containing the command to build the shared library

`custom_logging_functions.temp_custom_logging_dir()`

Function to return the path the custom logging library shared object and script will be written to.

Makes use of `os.makedirs exist_ok` which requires python 3.2+

## 3.2 functions module

`functions.create_arg_string(arg_dict)`

Function that creates the arg string

`functions.get_arg_keys()`

Function that return the list of possible keys to give in the arg string

`functions.get_defaults()`

Function that calls the binaryc get args function and cast it into a dictionary All the values are strings

`functions.load_logfile(logfile)`

`functions.parse_output(output, selected_header)`

Function that parses output of binary\_c:

This function works in two cases: if the caught line contains output like ‘example\_header time=12.32 mass=0.94 ..’ or if the line contains output like ‘example\_header 12.32 0.94’

You can give a ‘selected\_header’ to catch any line that starts with that. Then the values will be put into a dictionary.

TODO: Think about exporting to numpy array or pandas instead of a defaultdict

`functions.run_system(**kwargs)`

Wrapper to run a system with settings

This function determines which underlying python-c api function will be called based upon the arguments that are passed via kwargs.

- if `custom_logging_code` or `custom_logging_dict` is included in the kwargs then it will
- if

`functions.run_system_with_log(**kwargs)`

Wrapper to run a system with settings AND logs the files to a designated place defined by the `log_filename` parameter.

## INDICES AND TABLES

- genindex
- modindex
- search



## PYTHON MODULE INDEX

### **c**

`custom_logging_functions`, 5

### **e**

`examples.examples`, 3

### **f**

`functions`, 6



- A**
- autogen\_C\_logging\_code() (in module *custom\_logging\_functions*), 5
- B**
- binary\_c\_log\_code() (in module *custom\_logging\_functions*), 5
- binary\_c\_write\_log\_code() (in module *custom\_logging\_functions*), 5
- C**
- compile\_shared\_lib() (in module *custom\_logging\_functions*), 5
- create\_and\_load\_logging\_function() (in module *custom\_logging\_functions*), 5
- create\_arg\_string() (in module *functions*), 6
- custom\_logging\_functions (module), 5
- E**
- examples.examples (module), 3
- F**
- from\_binary\_c\_config() (in module *custom\_logging\_functions*), 5
- functions (module), 6
- G**
- get\_arg\_keys() (in module *functions*), 6
- get\_defaults() (in module *functions*), 6
- L**
- load\_logfile() (in module *functions*), 6
- P**
- parse\_output() (in module *functions*), 6
- R**
- return\_compilation\_dict() (in module *custom\_logging\_functions*), 5
- run\_example\_binary() (in module *examples.examples*), 3
- run\_example\_binary\_with\_custom\_logging() (in module *examples.examples*), 3
- run\_example\_binary\_with\_run\_system() (in module *examples.examples*), 3
- run\_example\_binary\_with\_writing\_logfile() (in module *examples.examples*), 3
- run\_system() (in module *functions*), 6
- run\_system\_with\_log() (in module *functions*), 6
- T**
- temp\_custom\_logging\_dir() (in module *custom\_logging\_functions*), 5