GPython Documentation

Release 0.8a

javabird25 (Protocs)

Sep 16, 2018

For Users:

Experimental project which strives to provide the possibility to create addons for Garry's Mod with Python 3 programming language.

CHAPTER 1

Lua Reference

You can interoperate with Python from Lua, for example, run arbitrary Python code. All interop functions are in the `py` global table.

`py.`**`Exec`**(*code*)

Executes a string of Python code.

```
py.Exec('print("HELLO".capitalize())')  -- Will print "Hello" to the console
```

CHAPTER 2

Building

2.1 Requirements for building

1. Cython
2. Visual Studio 2017

2.2 Instructions

1. Copy `lua_launcher\gpython_launcher` directory to `garrysmod\addons` directory.
2. Open command prompt, `cd` to `python_extensions` directory and run `setup.py build_ext --inplace`.
3. Move all files with `.pyd` extension in `python_extensions` directory to `garrysmod\gpython` directory.
4. Copy `python_extensions\gmod` directory to `garrysmod\gpython\gmod` (so there is a bunch of `.py` files in `garrysmod\gpython\gmod` directory).
5. Open `GPython.sln` with Visual Studio and build the solution.
6. Move `gmsv_gpython_win32.dll` and `gmcl_gpython_win32.dll` from `bin_modules\build` directory to `garrysmod\lua\bin` directory.
7. Move `gpython.dll` to Garry's Mod's root directory (where `hl2.exe` resides).

Final directory structure should looks like this:

```
...\SteamApps\GarrysMod\ ── hl2.exe
                          ├─ ...
                          ├─ gpython.dll
                          ├─ garrysmod\ ── addons\ ─── gpython_launcher\ ── addon.json
                          │              │                                  └─ lua\
↪──────── ...
```

(continues on next page)

(continued from previous page)

```
                          │              └─ lua\ ──────── bin\ ──────────────── gmsv_
↪gpython_win32.dll
                          │                                                    └─ gmcl_
↪gpython_win32.dll
                          └─ gpython\ ─── luastack.cpXX-win32.pyd
                                         └─ gmod\ ──────────────────────────── __init__.
↪py
                                                                               ├─ lua.py
                                                                               ├─ ...
                                                                             ...
```

See also:

Building documentation

CHAPTER 3

Building documentation

3.1 Requirements

1. Spxinx
2. CorLab theme

3.2 Instructions

1. Make sure that Sphinx is properly installed and built.
2. Just run `make html` in a command prompt in `docs\` directory.

Output is in `docs\_build\` directory.

CHAPTER 4

How GPython works

Let's suppose we have a simpliest "Hello World" GPython addon:

```
addons\ ── example\ ── addon.json
        ...           ├─ python\ ─── __autorun__\ ─── __init__.py
```

`__init__.py`:

```
from gmod import *

# Greeting the first player
print('Hello, ' + Player(1).nick + '!')
```

4.1 1. Lua Launcher

GPython's Lua launcher is a regular Lua addon. When it's loaded by Garry's Mod's addon system, the launcher activates `gmcl_gpython_win32.dll` and `gmsv_gpython_win32.dll`:

```
require 'gpython'
```

`gmcl_gpython_win32.dll` is for the *client* and `gmsv_gpython_win32.dll` is for the *server*.

4.2 2. Realms' DLLs

These two DLLs call `gpython_run()` in `gpython.dll`.

4.3 3. Main GPython DLL: `gpython.dll`

`gpython_run()` does these preparation operations:

4.3.1 Server

1. Adds `luastack` module to the builtin initialization table.
2. Initializes Python interpreter.
3. Appends `garrysmod\gpython\` to `sys.path`.
4. Calls `setup()` in `luastack` thus setting the global lua stack pointer and setting `luastack.IN_GMOD` to `True`.
5. Redirects I/O to Garry's Mod console with `gmod.streams`.
6. Adds *Lua2Python interoperability functions* (using Python from Lua).
7. Scans `addons\` directory for GPython addons and runs their code.

4.3.2 Client

Client's routine is the same as server's except the step **2**.

Instead of initializing Python again, a subinterpreter is created and swapped to.

4.4 4. GPython wrappers

`Player` is just a wrapper over the corresponding Lua functions, as much as many other GPython services.

In this example, `Player(1)` creates an object that wraps `Player` Lua class. `nick` is a property that gets players' nicknames using Player:Nick() function.

This property looks like this:

```
@property
def nick(self):
    return str(self._player_luaobj['Nick']())
```

The `Player` internally uses `gmod.lua.LuaObject` to work with players.

4.5 5. `gmod.lua` module

`gmod.lua` module is itself a wrapper over `luastack` module. `gmod.lua` simplifies the interoperability with Lua by providing `LuaObject` class and `G` singleton.

`LuaObject` internally uses the *luastack module*.

4.6 6. `luastack` module

luastack module manipulates the Lua stack. Lua stack pointer is previously set by the C++ module.

And that's it, our GPython addon is initialized. For me, `Hello, Protocs!` will be printed to console.

CHAPTER 5

Internal modules

These modules are not intended to be used by addon developers.

5.1 `luastack`: Lua Stack manipulation

Note:

`luastack` is not a part of the `gmod` package. So, instead of:

```
from gmod.luastack import *
```

or:

```
from gmod import *
# Trying to use luastack
```

you should use:

```
from luastack import *
# Using luastack
```

This module contains functions for manipulating the Lua stack. The lowest level of Garry's Mod Lua interoperability.

5.1.1 Lua Stack description

Lua uses a virtual stack to pass values to and from C. Each element in this stack represents a Lua value (`nil`, number, string, etc.).

For convenience, most query operations in the API do not follow a strict stack discipline. Instead, they can refer to any element in the stack by using an index:

- A positive index represents an absolute stack position (starting at **1**)
- A negative index represents an offset relative to the top of the stack.

More specifically, if the stack has **n** elements, then index **1** represents the first element (that is, the element that was pushed onto the stack first) and index **n** represents the last element; index **-1** also represents the last element (that is, the element at the top) and index **-n** represents the first element. We say that an index is valid if it lies between **1** and the stack top (that is, if **1 abs(index) top**).

IN_GMOD
Boolean constant which is `True` if GPython system is running during a Garry's Mod session. This constant is used in GPython libraries for preventing some code from executing when generating documentation.

ILuaBase ***lua**
Pointer to Lua Stack C++ object. Not available to Python.

Being set by *setup()*.

void **setup**(ILuaBase **lua*)
Sets the global lua pointer and *IN_GMOD*.

5.1.2 Stack manipulation

top()
Returns the index of the top element in the stack.

Because indices start at **1**, this result is equal to the number of elements in the stack (and so **0** means an empty stack).

equal(*a: int*, *b: int*) → bool
Returns `True` if the two values in acceptable indices `a` and `b` are equal, following the semantics of the Lua == operator (that is, may call metamethods). Otherwise returns `False`. Also returns `False` if any of the indices is non valid.

raw_equal(*a: int*, *b: int*) → bool
Returns `True` if the two values in acceptable indices `a` and `b` are primitively equal (that is, without calling metamethods). Otherwise returns `False`. Also returns `False` if any of the indices are non valid.

insert(*destination_index: int*)
Moves the top element into the given valid index, shifting up the elements above this index to open space.

throw_error(*message: bytes*)
Raises a Lua error with the given `bytes` message.

It also adds at the beginning of the message the file name and the line number where the error occurred, if this information is available.

create_table() → None
Creates a new empty table and pushes it onto the stack.

push(*stack_pos: int*)

push_special(*type: int*)
Pushes a special value onto the stack. Use *Special* enum.

push_nil()
Pushes a `nil` value onto the stack.

push_number(*num*)
Pushes a number `num` onto the stack.

push_string(*val: bytes*)
Pushes a `bytes` object onto the stack.

You can push `str` like this:

```
s = '...'
push_string(s.encode())
```

push_bool(*val: bool*)
Pushes a boolean value `val` onto the stack.

pop(*amt: int = 1*) → None
Pops `amt` elements from the stack.

clear()
Clears the stack (pops all values).

get_table(*stack_pos: int*) → None
Pushes onto the stack the value `t[k]`, where `t` is the value at the given valid index `stack_pos` and `k` is the value at the top of the stack.

This function pops the key from the stack (putting the resulting value in its place).

get_field(*stack_pos: int*, *name: bytes*) → None
Pushes onto the stack the value `t[name]`, where `t` is the value at the given valid index `stack_pos`.

set_table(*stack_pos: int*)
Does the equivalent to `t[k] = v`, where `t` is the value at the given valid index `stack_pos`, `v` is the value at the top of the stack, and `k` is the value just below the top.

This function pops both the key and the value from the stack.

set_field(*stack_pos: int*, *name: bytes*)
Does the equivalent to `t[k] = v`, where `t` is the value at the given valid index `stack_pos` and `v` is the value at the top of the stack.

This function pops the value from the stack.

call(*args: int*, *results: int*)
Calls a function.

To call a function you must use the following protocol:

1. The function to be called is pushed onto the stack
2. The arguments to the function are pushed in direct order; that is, the first argument is pushed first.
3. You call this function; `args` is the number of arguments that you pushed onto the stack.

All arguments and the function value are popped from the stack when the function is called. The function results are pushed onto the stack when the function returns. The number of results is adjusted to `results`. The function results are pushed onto the stack in direct order (the first result is pushed first), so that after the call the last result is on the top of the stack.

The following example shows how the host program can do the equivalent to this Lua code:

```
a = f("how", t.x, 14)
```

Here it is with GPython's luastack:

```
push_special(Special.GLOBAL)
get_field(1, "f")  # function to be called
push_string("how")  # 1st argument
```

(continues on next page)

(continued from previous page)

```
get_field(1, "t")  # table to be indexed
get_field(-1, "x")  # push result of t.x (2nd arg)
remove(-2)  # remove 't' from the stack
push_number(14)  # 3rd argument
call(3, 1)  # call 'f' with 3 arguments and 1 result
set_field(1, "a")  # set global 'a'
```

Note that the code above is "balanced": at its end, the stack is back to its original configuration. This is considered good programming practice.

get_string(*stack_pos: int = -1*) → bytes
Returns the `bytes` value of a string item at `stack_pos` index of the stack.

Negative values can be used for indexing the stack from top.

get_number(*stack_pos: int = -1*) → float
Returns the `float` value of a number item at `stack_pos` index of the stack.

Negative values can be used for indexing the stack from top.

get_bool(*stack_pos: int = -1*) → bool
Returns the boolean value of a boolean item at `stack_pos` index of the stack.

Negative values can be used for indexing the stack from top.

create_ref() → int
Saves the value at the top of the stack to a reference, pops it and returns the reference ID.

free_ref(*ref: int*)
Frees the reference with ID `ref`.

push_ref(*ref: int*)
Pushes the value saved in the reference with ID `ref` onto the stack.

get_type(*stack_pos: int*) → ValueType
Returns *`ValueType`* enum member which corresponds to the type of the value at index `stack_pos` of the stack.

get_type_name(*type: ValueType*) → str
Returns a lowercase string representation of `type` type.

is_type(*stack_pos: int, type: ValueType*) → bool
Returns `True` if the value's type at index `stack_pos` is the same as `type` argument.

5.1.3 Special values

class Special(*enum.enum*)
Enum of special values. The only practically usable value is *`Special.GLOBAL`*.

Special values can be pushed with *`push_special()`*.

GLOBAL
Represents the global Lua namespace (`_G`).

ENVIRONMENT
Represents the environment table.

REGISTRY
Represents the registry table. More data can be found in Official Lua documentation.

5.1.4 Value types

class ValueType(*enum.enum*)

Enum of Lua value types.

```
NIL,
BOOL,
LIGHTUSERDATA,
NUMBER,
STRING,
TABLE,
FUNCTION,
USERDATA,
THREAD,

# UserData
ENTITY,
VECTOR,
ANGLE,
PHYSOBJ,
SAVE,
RESTORE,
DAMAGEINFO,
EFFECTDATA,
MOVEDATA,
RECIPIENTFILTER,
USERCMD,
SCRIPTEDVEHICLE,

# Client Only
MATERIAL,
PANEL,
PARTICLE,
PARTICLEEMITTER,
TEXTURE,
USERMSG,

CONVAR,
IMESH,
MATRIX,
SOUND,
PIXELVISHANDLE,
DLIGHT,
VIDEO,
FILE
```

5.2 `streams`: I/O to Garry's Mod

CHAPTER 6

`lua`: Lua interoperability

CHAPTER 7

`net`: communication between Client and Server

CHAPTER 8

`player`: player utilities

CHAPTER 9

`entity`: entity utilities

CHAPTER 10

`realms`: determining the current realm

- modindex
- search

CHAPTER 11

Useful links

- Contribute on GitHub

Index