Creating a module

How to create a MAQAO module

Version 1.0

Contents

..... 0
1 Prequisites 2
2 Creating a Module 3
3 Running The Module 5
4 Using parameters 7
5 Aliases 9
6 Man page 11
7 Example 11
7.1 Use several source files 11
7.2 The Help Class 12
8 More performance using C code 14

1 Prequisites

MAQAO tutorials: MAQAO Lua API

2 Creating a Module

MAQAO has an option to generate a module skeleton. The command to generate the skeleton is:

```
$> maqao --module-skeleton <module_name> [output=<path>]
```

The command needs the module name. The output option is optional and can be used to specify where the module will be created. The default path is the current directory. The usage of the command is presented in Figure 1.

```
[...]$ maqao --module-skeleton basic_stats

Skeleton of module basic_stats has been created in
/home/user/Desktop/basic_stats

[...]$
```

Figure 1: Generate a module using the MAQAO command

Once the command is ran, a new directory with the module name is created. This directory is the module. It contains a Lua file named after it and another file called help.lua. The Lua file is the main file of the module. It contains the default code described by Figure 2. The help.lua file declares a function used to load the module help. Help is described in a section 6.2.

N.B.: Do not modify this file name because the module will not work anymore.

```
( 1)module ("basic_stats", package.seeall)
( 2)
( 3)require ("basic_stats.help")
( 4)
( 5)--- Entry point of the module basic_stats
( 6)--@param args A table of parameters
( 7)--@param aproject An existing project
( 8)function basic_stats:basic_stats_launch (args, aproject)
( 9)  local help = basic_stats:init_help()
(10)  -- Print version if -v or --version are passed in parameters
(11)  if args.v == true or args.version == true then
(12)    help:print_version (false)
(13)  end
(14)  -- Print help if -h or --help are passed in parameters
(15)  if args.h == true or args.help == true then
(16)    help:print_help (false)
(17)  end
(18)  -- If help or version have been displayed, exit
(19)  if args.h == true or args.help == true
(20)  or args.v == true or args.version == true then
(21)    os.exit (0)
(22)  end
(23)  --Insert your code here
(24)end
```

Figure 2: Generated main file contents

Line (1) of Figure 2 is used to register the current module to MAQAO. The module's entry point (main) function is declared at line (8).

N.B.: Do not change its name.

The function takes two parameters:

- args: a Lua Table object which contains parameters passed when the module will be ran. Its structure will be described later.
- project: an initialized project object needed by MAQAO. It is named after the module's name. Note that other projects can be created and used. This is only a default project (container) because developers rarely want to set a project name.

3 Running The Module

At this point, the module is created and can be used. It is necessary to specify to MAQAO where your module is. You can use the command below to verify if your module is properly registered:

```
$> maqao --list-modules
```

It will list existing modules. An example of the output of this command is shown in Figure 3.

```
[...]$ maqao --list-modules

  MODULE              ALIAS               LOCATION
================================================
  madras              madras              built-in
  mil                 instrument          built-in
  analyze             analyze             built-in
  stan                                    built-in
  perfeval            perf                built-in
  grouping                                built-in
  basic_stats                             /home/user/Desktop
[...]$
```

Figure 3: Listing available modules

Built-in modules are modules available in the binary. The alias column gives a list of shortcuts which be used to run the module. Aliases are described in section 5. To specify to MAQAO where to look for modules, the environment variable MAQAO_PLUGIN_PATH has to be set (with paths to your modules). Multiple paths can be specified using ':' between each path, such as other Unix environment variables. By default, MAQAO looks for modules in the current folder. That is why basic_stats module appears in the module list.

A module can be launched in the following way:

```

```

Figure 4 shows how to run a module located in the current directory whereas the example in Figure 5 shows how to run a module using MAQAO_PLUGIN_PATH environment variable.

```
[...]$ maqao basic_stats
[...]$
```

Figure 4: Run a module located in current directory

```
[...]$ export MAQAO_PLUGIN_PATH=$PWD
[...]$ maqao basic_stats
[...]$
```

Figure 5: Run a module specifying MAQAO_PLUGIN_PATH

Since our module does nothing, nothing is output in Figure 4 and 5. After adding the line:

```
      print ("Hello World !")
```

in the main function at line (26) and running again the module we get the output shown in Figure 6.

```
[...]$ maqao basic_stats
Hello World !
[...]$
```

Figure 6: Run a module displaying “Hello world”

4 Using parameters

Parameters are passed to the module using the first parameter of the main function. This parameter is a Table object using the option name as key and a string as value. To display passed parameters, the following line can be added to the main function code:

```
args:tostring()
```

Figure 7 shows default parameters passed to the module.

```
[...]$ maqao basic_stats
Hello World !
Table =
{
        ["lua_script"] = false;
        ["batch"] = false;
        ["_alias"] = "basic_stats";
};
[...]$
```

Figure 7: Calling the module without other parameter

There are three ways to pass parameters to a module:

- short option: -<sopt>[=val]
- long option: --<lopt>[=val]
- a third type of option: <opt>=<val>

When an option has no value, the boolean value "true" is used as default value. Figure 8 presents the three ways to pass parameters to a module.

```
[...]$ maqao basic_stats --longopt1 --longopt2=val2 -o opt3=val3
Hello World !
Table =
{
        ["longopt1"] = true;
        ["o"] = true;
        ["_alias"] = "basic_stats";
        ["lua_script"] = false;
        ["batch"] = false;
        ["longopt2"] = "val2";
        ["opt3"] = "val3";
};
[...]$
```

Figure 8: Calling the module with all supported format off parameters

There is an additional mean to pass options that will not be interpreted by MAQAO. The substring "--" can be used in the command line to specify that everything after it must be kept verbatim in a single string. The string is saved in the option "_bin_params". This is shown by figure 9.

```
[...]$ maqao basic_stats --longopt1 -- --longopt1=v1 -opt
Hello World !
Table =
{
        ["longopt1"] = true;
        ["_alias"] = "basic_stats";
        ["_bin_params"] = " --longopt1=v1 -opt";
        ["lua_script"] = false;
        ["batch"] = false;
};
[...]$
end
```

Figure 9: Calling the module using the special token “--"

It is very usefull when you want to call an external program from your module and want to pass to it options that we don’t want MAQAO to interprete.

5 Aliases

Aliases are shortcuts which can be used to simplify the usage of modules. Aliases can be defined in a file called "alias" located in the module directory. There are two kinds of aliases:

- short alias: a word replacing the default name <module_name>. The syntax of short aliases is a single word on a line matching the regular expression "^[a-zA-Z0-9_\-]+$". Only one short alias can be defined per module.
- long alias: a word replacing a list of options, including the module name <module_name> option. The syntax of long aliases is a short alias, a ':' character then a list of options which are always passed when the long alias is used.

Figure 10 depicts an example of an alias file. Figure 11 and 12 show different ways to run a module using aliases.

```
(1) bstats
(2) bstats_text : --format=txt
```

Figure 10: Example of alias file for a module

```
[...]$ maqao bstats
Hello World !
Table =
{
        ["lua_script"] = false;
        ["batch"] = false;
        ["_alias"] = "bstats";
};
[...]$
```

Figure 11: Run the module using the short alias defined at line (1)

```
[...]$ maqao bstats_text --longopt
Hello World !
Table =
{
        ["format"] = "txt";
        ["lua_script"] = false;
        ["batch"] = false;
        ["longopt"] = true;
        ["_alias"] = "bstats_text";
};
[...]$
```

Figure 12: Run the module using the long alias defined at line (2) and adding manually an option

```
[...]$ maqao --list-modules

  MODULE              ALIAS                LOCATION
=========================================================
  madras              madras               built-in
  mil2                instrument           built-in
  analyze             analyze              built-in
  stan                                     built-in
  perfeval            perf                 built-in
  mil                 mil1                 built-in
  grouping                                 built-in
  basic_stats         bstats               /home/user/Desktop
                      bstats_text          --format=txt
[...]$
```

Figure 13: List modules to get their aliases

Figure 13 presents the output of the –list-modules command with a module using aliases. The first line (for a given module) presents the short alias and the location of the module. Next lines described long aliases and which options are defined by these aliases in the location column.

6 Man page

MAQAO implements the option --generate-man to generate the man page of a module. The command is presented by figure 14.

```
[...]$ maqao <module> --generate-man
```

Figure 14: How to generate a module man page

MAQAO uses the content of the Help module to produce the man page. The produced file is generated in current directory, but the path can be changed using to --output option. The produced file is called maqao-<module name>.1.

7 Example

The tutorial's associated download basic_stats.tar.gz contains a module performing basic statistics on a given binary. It is composed of four source files listed in figure 15 and defines some aliases presented in Figure 11.

```
basic_stats.lua
analyze.lua
output/csv.lua
output/text.lua
```

Figure 15: List of source files in the module basic_stats

7.1 Use several source files

In order to simplify the source code, a module can be split into several source files. Lua has its own concept of module. Each file must begin with a declaration of the current Lua module, using the keyword module. It is shown by the line (1) in Figure 2. The first parameter of module function is the relative path of the file from the module root and without the .lua file extension.

The main file is automatically loaded by MAQAO if the environment variable MAQAO_PLUGIN_PATH has been set properly (see section 2 for more information). However, when there are several other source files, they must be explicitly loaded. To this end, the Lua function "*require*" must be used. The parameter of the require function is the path set in the module declaration of the file to load. In our example we define three additional source file as shown in Figure 16.

```
-- Load other source files
require (“basic_stats.output.csv”)
require (“basic_stats.output.text”)
require (“basic_stats.analyze”)
```

Figure 16: Loading source files

7.2 The Help Class

In order to standardize help displaying in MAQAO, a class "Help" has been implemented. It defines several functions that must be used to display the module manual. A default Help object is created at module generation in file help.lua presented by figure 17. Figure 18 is the code used in the module to loading the Help object for the module. Figure 19 presents a way to call the help if it is asked trough parameters.

```
module ("basic_stats.help", package.seeall)

function basic_stats:init_help()
  local help = Help:new();
  help:set_name ("basic_stats");
  -- Add your code here
  Utils:load_common_help_options (help)
  return help;
end
```

Figure 17: Content of help.lua at module generation

```
-- Generate the help content
local help = Help:new()
help:set_name ("maqao-basic_stats")
help:set_usage ("maqao module=basic_stats bin=<binary> [...]")
help:set_description ("This module computes basic statistics in a
given binary")
help:add_option ("bin", nil, "<binary>", false, "Select the binary
to analyze")
help:add_option ("format", "f", "<value>", false,
         "Select the display format. Available values are
\"text\", \"csv\"")
Utils:load_common_help_options (help)
```

Figure 18: Example of a code used to load Help in a module

```
-- Print version if -v or --version are passed in parameters
if args.v == true or args.version == true then
   help:print_version (false)
end
-- Print help if -h or --help are passed in parameters
if args.h == true or args.help == true then
   help:print_help (false)
end
-- If help or version have been displayed, exit
if args.h == true or args.help == true
or args.v == true or args.version == true then
   os.exit (0)
end
```

Figure 19: Example of a code handling help and version parameters

8 More performance using C code

It is possible to call C code in a modules wrote in Lua. This feature will be covered in another upcoming tutorial.