ARMY RESEARCH LABORATORY

Simple Plugin Methodology in Python

by Ralph P Ritchey and Travis W Parker

ARL-CR-0743 **August 2014**

Prepared by

ICF International
7125 Thomas Edison Drive Suite 100
Columbia, MD 21046

Under contract

W911QX-14-F-0020

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

Army Research Laboratory
Adelphi, MD 20783-1138

ARL-CR-0743 **August 2014**

Simple Plugin Methodology in Python

Ralph P Ritchey and Travis W Parker
Computational and Information Sciences Directorate, ARL

Prepared by

ICF International
7125 Thomas Edison Drive Suite 100
Columbia, MD 21046

Under contract

W911QX-14-F-0020

REPORT DOCUMENTATION PAGE		Form Approved OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
September 2014	Final	

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Simple Plugin Methodology in Python	W911QX-14-F-0020
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)	5d. PROJECT NUMBER
Ralph P Ritchey and Travis W Parker	5e. TASK NUMBER
	5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
ICF International 7125 Thomas Edison Drive Suite 100 Columbia, MD 21046	

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
US Army Research Laboratory ATTN: RDRL-CIN-D 2800 Powder Mill Road Adelphi, MD 20783-1138	ARL-CR-0743
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES

Government POC: Robert Reschly

14. ABSTRACT

This report provides a simple plugin methodology using the Python programming language. The methodology supports dynamic loading of plugins at execution time, allowing the user to select which plugin to use from the command line.

15. SUBJECT TERMS

Python, plugin, methodology, dynamic framework

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
A. Report	b. ABSTRACT	c. THIS PAGE			Ralph P Ritchey
Unclassified	Unclassified	Unclassified	UU	18	19b. TELEPHONE NUMBER (*Include area code*) (301) 394-0780

Contents

INTENTIONALLY LEFT BLANK.

1. Introduction

While developing a simulator in Python that would allow iterative evolution of portions of the code for experimental purposes during a research and development project, we determined that a plugin methodology would facilitate an easier, more scalable, and maintainable approach. This capability also allows fast, automated repetition of experiments based on command line arguments passed to the simulator.

This report describes the implementation that was settled upon in its basic, stripped-down sample script that can be adapted by others for their use. An example where this methodology could be leveraged is in a Python script where a key function or method needs to be updated regularly by the developer over time. As newer updated versions of the plugin are created, the older versions can be easily maintained for comparison or regression testing. Suggestions for further improvements that others may find useful are supplied at the end of this report.

2. Configuration Used

The following list is the operating system version and Python version used while the plugin methodology was being developed. Due to operating system and application programming interface (API) call variances, slight adjustments may be needed depending on the local development environment:

- Operating System: Red Hat Enterprise Linux (RHEL) version 6.5
- Python Version: 2.7.6
- Dell Optiplex 960
  - 8 GB Memory
  - Intel Core2 CPU
    - Quad Core

3. The Approach

With the primary focus for the project being on the calculations performed on the data and the results, the desire was to keep the plugin capability as simple as possible. While there are numerous plugin architectures available,[1] many contained more overhead, were far more

sophisticated, and required more external dependencies[2] than we desired or they didn't quite meet our exact use case.[3,4]

Our implementation for a simplified plugin architecture meets our key requirements:

- *No external dependencies*: The only dependencies that are imported are several extra libraries already supplied natively by a base Python installation.
- *Simple*: Plugins of a specific type or capability are grouped within the same subdirectory. When loaded, each plugin is easily accessed via a dictionary object by a unique name. Beyond API requirements specific to the plugin's intended use and a help method, there are no implementation requirements that are required or enforced by the plugin implementation or other frameworks and libraries.
- *User friendly*: Due to the ability to dynamically add and remove plugins, the help capability (`-h` and `--help` command line options) should dynamically update as well. This allows the user to quickly and easily determine which plugins are available and what their intended use is.

4. Directory Structure

The directory structure used consists of two levels, a primary level where the core (main) Python script resides and a secondary level consisting of subdirectories containing the plugins. The plugin loader as implemented assumes the subdirectories containing the plugins are contained within the same directory as the main Python script itself:

```
/simulator
|-- calculators
|   |-- calcMethod1.py
|   `-- calcMethod2.py
|-- outputs
|   |-- graph.py
|   `-- text.py
`-- sim_main.py
```

For our use case, we are grouping plugins that provide the same type of capability under specific subdirectories. In addition to keeping the plugins organized, this facilitates loading the related plugins into the same dictionary object for easy access within the simulator.

5. Plugin Structure

The plugin code is simple—each is a self-contained Python object. The class name (`simPlugin`) is the same throughout all plugins, so they can be generically loaded without writing code specific to a plugin type or unique API. The following is a sample of the skeleton plugin code:

```
class simPlugin(object):
    state = ""

    def __init__(self):
        self.state = "Initialized"

    def print_help(self):
        return "Name:  text\n\tPrints calculation results as text output."

    def output_results(self, parameter1, parameter2):
        """
        parameters1:  Standard first parameter passed to all output type
plugins....
        parameters2:  Standard second parameter passed to all output type
plugins....
        """
        print "Code for outputting text results goes here...."
```

In the skeleton plugin code shown above, the three required methods are supplied. The `__init__` method is used when the plugin is first instantiated and is a standard object constructor Python provides, the same as many other programming and scripting languages provide. This provides an opportunity for the plugin to initialize any internal settings or execute any unique code that plugin must execute when first instantiated.

The `print_help` method is required by our implementation as it is automatically executed when the built-in help text is requested by the user. The method should clearly indicate the name of the plugin (which the user will use to select that plugin when using command line options) and what it does.

In the case of our output plugins, the `output_results` method is the standard API call to trigger the code for generating output by the output plugin. All of our output plugins implement this method as the standard API call for generating output. Other plugin types, such as calculation plugins, use a differently named method—but it is consistent throughout all calculation plugins (`perform_calculation`).

6. Plugin Methodology Core

The full source code for the script using the simple plugin methodology discussed in this report is located in the Appendix. Subsections of the script are presented here for easy reference while portions of the script specific to the methodology implementation are discussed. For questions regarding portions of the script specific to the Python scripting language and its natively provided API calls, the reader is encouraged to refer to the authoritative Python language documentation.[5]

The primary function to the plugin methodology is contained in the main script and is called `loadPlugins`. This function requires two parameters to be passed in, the first containing the name of the subdirectory for the plugin(s) to be loaded and the second providing the name of the global dictionary they should be loaded into. The following is an example of this function:

```
def loadPlugins(directory, whichDictionary):
    myPlugIns = glob.glob(directory + "/*.py")

    for file in myPlugIns:
        # Extract just the first part of the .py file name.
        name = file.split("/")[1].split(".")[0]
        print "Loading plugin: " + name

        # Dynamically set the PYTHONPATH so the user doesn't have to.
        # It assumes the plugins are contained in subdirectories where
        #the main file lives.
        path = os.path.dirname(sys.argv[0])
        if len(path) == 0:
            path = "."
        sys.path.append(path + "/" + directory)

        # Import the plugin module temporarily long enough to
        # instantiate an object which is stored in a globally
        # accessible dictionary.
        tempModule = __import__(name)
        whichDictionary[name]  = tempModule.simPlugin()
```

After obtaining the names of all plugin files contained in the subdirectory, the function loops through each one creating a dictionary entry accessible by the root portion of the plugin's filename (key) and containing an instantiated plugin object as its element (value).

The `availablePlugins` function plays an important role as it is used to dynamically print the list of available plugins for each available type with a brief description when the user uses the command line help option (`-h` or `--help`). This dynamic ability allows the help output to be adjusted automatically as plugins are added or removed without requiring the source code contained in the main script.

The last piece of important code contained in the main script file are the examples showing the calculation and output plugins being dynamically called based on the user's command line arguments:

```
# Based on the user selected plug-in, perform the calculation.
if (args.calculation) and (args.calculation in calculation_objects):
    calculation_objects[args.calculation].perform_calculation("dummyValue")

# Based on the user selected plug-in, output the results.
if (args.output) and (args.output in output_objects):
    output_objects[args.output].output_results("dummyValue", "dummyValue")
```

7. Suggested Changes Depending on Use

While the source code documented in this report serves our specific use case, there are several suggested changes that, while still keeping the implementation simple, may be beneficial to others:

- *Single plugin dictionary*: We chose to explicitly separate our two plugin types (calculation and output) into two separate subdirectories and maintain them while the script is executing in separate dictionaries. By using a multi-dimensional dictionary, all plugins could be stored within the same dictionary by using the plugin type as the primary key and the plugin name as a secondary key for accessing the instantiated plugin object.
- *Instantiated plugin API parameters*: For our implementation we assumed all plugins of the same type will have exactly the same parameters for all standardized API calls. If this does not fit the use case, one methodology would be to store all parameters that are to be passed in to plugin in a dictionary. When passed to the plugin, it would use the dictionary to retrieve the specific parameters it is interested in, ignoring the extra, unused parameters. Another option would be to query the plugin's API call to determine which parameters it requires and then pass only those specific parameters. Each of these methodologies introduces different levels of complexity, and it would be up to the implementer to determine which would be best for their particular situation.
- *Only instantiating used plugins*: The implementation here loads all available plugins whether they are used or not. In situations where there numerous plugins or they are large in size it may be beneficial memory and processing wise to only load the plugin(s) that are actually going to be used. This would be possible by loading the plugins after processing the user's command line arguments.
- *Security*: Care should be taken if the primary script is executed using elevated privileges to ensure users are unable to introduce malicious plugins that are then automatically loaded and executed.

8. Conclusion

The plugin mechanism introduced in this report provides an easy method for Python software developers to dynamically add or remove functionality for their scripts. By limiting the developers focus to creating and updating plugins, no or very minimal changes are required to the broader source code base, greatly reducing the introduction of bugs or unintended side effects.

As used in a simulation script for our research project, the plugin methodology has allowed us to easily add in new calculation engine plugins as the mathematics behind the calculations evolved over time. The user or researcher can now dynamically select the calculation engine to be used based on command line arguments.

9. References

1. Hart WE, William E. Hart's Blog. 26 January 2009 [Accessed 25 June 2014]. [Online]. Available: http://wehart.blogspot.com/2009/01/python-plugin-frameworks.html.
2. Alchin M. A Simple Plugin Framework. 10 January 2008 [Accessed 25 June 2014]. [Online]. Available: martyalchin.com/2008/jan/10/simple-plugin-framework/.
3. Lews J. One Hour Hacks - A Simple Python Plugin Framework. 24 April 2013 [Accessed 25 June 2014]. [Online]. Available: http://onehourhacks.blogspot.com/2013/04/a-simple-python-plugin-framework.html.
4. MiJyn. lkubuntu - Writing a python plugin API/Architecture. 2 October 2012 [Accessed 25 June 2014]. [Online]. Available: http://lkubuntu.wordpress.com/2012/10/02/writing-a-python-plugin-api/.
5. Python Software Foundation, Python v2.7.7 documentation. 28 June 2014 [Accessed 30 June 2014]. [Online]. Available: https://docs.python.org/2.7/.

INTENTIONALLY LEFT BLANK.

Appendix. Main Script Source Code: sim_main.py

The following is the main script code, sim_main.py:

```
import glob      # Imported for file directory listing capabilities.
import sys       # Imported for access to command line parameters
import os        # Imported for file path manipulation capabilities.
import argparse  # Imported for handling command line arguments.

calculation_objects = {}
output_objects = {}

def loadPlugins(directory, whichDictionary):
    myPlugIns = glob.glob(directory + "/*.py")

    for file in myPlugIns:
        # Extract just the first part of the .py file name.
        name = file.split("/")[1].split(".")[0]
        print "Loading plugin: " + name

        # Dynamically set the PYTHONPATH so the user doesn't have to.  It
assumes
        # the plugins are contained in subdirectories where the main file
lives.
        path = os.path.dirname(sys.argv[0])
        if len(path) == 0:
            path = "."
        sys.path.append(path + "/" + directory)

        # Import the plugin module temporarily long enough to instantiate an
object
        # which is stored in a globally accessible dictionary.
        tempModule = __import__(name)
        whichDictionary[name]  = tempModule.simPlugin()

def availablePlugins():
    text = "Available 'calculation' plug-ins:\n"
    for i in calculation_objects:
        text += calculation_objects[i].print_help() + "\n"

    text += "\n\nAvailable 'output' plug-ins:\n"
    for i in output_objects:
        text += output_objects[i].print_help() + "\n"
    return text

if __name__ == "__main__":

    # Load plugins for performing the different steps in our calculation.
    loadPlugins("calculators", calculation_objects)
    loadPlugins("outputs", output_objects)

    parser =
argparse.ArgumentParser(formatter_class=argparse.RawDescriptionHelpFormatter,
epilog=availablePlugins())
    parser.add_argument("-c", "--calculation", help="The desired method to be
used for calculating the risk metric", type=str)
    parser.add_argument("-o", "--output", help="The desired method to be used
for outputting the risk metric calculation results", type=str)
```

```
    parser.add_argument("config_file", help="The configuration file to be
used containing the nodes/vulnerabilities")
    parser.parse_args()
    args = parser.parse_args()

    # Read in the configuration file to be used.
    print "Insert code here to read in configuration file..."

    # Based on the user selected plug-in, perform the calculation.
    if (args.calculation) and (args.calculation in calculation_objects):

calculation_objects[args.calculation].perform_calculation("dummyValue")

    # Based on the user selected plug-in, output the results.
    if (args.output) and (args.output in output_objects):
        output_objects[args.output].output_results("dummyValue",
"dummyValue")
```

1	DEFENSE TECH INFO CTR
(PDF)	ATTN DTIC OCA
2	US ARMY RSRCH LABORATORY
(PDF)	ATTN IMAL HRA MAIL & RECORDS MGMT
	ATTN RDRL CIO LL TECHL LIB
1	GOVT PRNTG OFC
(PDF)	ATTN A MALHOTRA
3	US ARMY RSRCH LAB
(PDF)	ATTN RDRL CIN D B RESCHLY
	ATTN RDRL CIN D R RITCHEY
	ATTN RDRL CIN D T PARKER