# lml

***Release 0.1.0***

**Oct 21, 2020**

# Contents

**Author** C.W.

**Source code** http://github.com/lml/lml.git

**Issues** http://github.com/lml/lml/issues



**Released** 0.1.0

**Generated** Oct 21, 2020

CHAPTER 1

# Introduction

**lml** seamlessly finds the lml based plugins from your current python environment but loads your plugins on demand. It is designed to support plugins that have external dependencies, especially bulky and/or memory hungry ones. lml provides the plugin management system only and the plugin interface is on your shoulder.

**lml** enabled applications helps your customers[1] in two ways:

1. Your customers could cherry-pick the plugins from pypi per python environment. They could remove a plugin using *pip uninstall* command.
2. Only the plugins used at runtime gets loaded into computer memory.

When you would use **lml** to refactor your existing code, it aims to flatten the complexity and to shrink the size of your bulky python library by distributing the similar functionalities across its plugins. However, you as the developer need to do the code refactoring by yourself and lml would lend you a hand.

[1] the end developers who uses your library and packages achieve their objectives.

CHAPTER 2

# Quick start

The following code tries to get you started quickly with **non-lazy** loading.

```
from lml.plugin import PluginInfo, PluginManager


@PluginInfo("cuisine", tags=["Portable Battery"])
class Boost(object):
    def make(self, food=None, **keywords):
        print("I can cook %s for robots" % food)


class CuisineManager(PluginManager):
    def __init__(self):
        PluginManager.__init__(self, "cuisine")

    def get_a_plugin(self, food_name=None, **keywords):
        return PluginManager.get_a_plugin(self, key=food_name, **keywords)


if __name__ == '__main__':
    manager = CuisineManager()
    chef = manager.get_a_plugin("Portable Battery")
    chef.make()
```

At a glance, above code simply replaces the Factory pattern should you write them without lml. What's not obvious is, that once you got hands-on with it, you can start work on how to do **lazy** loading.

## 2.1 Documentation

### 2.1.1 Design idea

The idea, to load the plugins later, originated from pyexcel project[1] which uses loosely coupled plugins to extend the main package to read more file formats. During its code growth, the code in pyexcel packages to manage the external and internal plugins becomes a independent library, lml.

Lml is similar to **Factories** in Zope Component Architecture[2]. It provides functionalities to discover, register and load lml based plugins. It cares how the meta data were written but it does NOT care how the plugin interface is written.

Simply, lml promises to load your external dependency when they are used, but only when you follow lazy-loading design principle below. Otherwise, lml does immediate import and takes away the developer's responsibility to manage plugin registry and discovery.

In terms of extensibility of your proud package, lml keeps the door open even if you use lml for immediate import. As a developer, you give the choice to other contributor to write up a plugin for your package. As long as the user would have installed community created extensions, lml will discover them and use them.

#### Plugin discovery

Prior to lml, three different ways of loading external plugins have been tried in pyexcel. namespace package[3] comes from Python 3 or pkgutil style in Python 2 and 3. It allows the developer to split a bigger packages into a smaller ones and publish them separately. sphinxcontrib[4] uses a typical namespace package based method. However, namespace package places a strict requirement on the module's *__init__.py*: nothing other than name space declaration should be present. It means no module level functions can be place there. This restriction forces the plugin to be driven by the main package but the plugin cannot use the main package as its own library to do specific things. So namespace package was ruled out.

The Flask extension management system was used early versions of pyexcel(=<0.21). This system manipulates sys.path so that your plugin package appears in the namespace of your main package. For example, there is a xls plugin called pyexcel-xls. To import it, you can use "import pyexcel.ext.xls". The shortcomings are:

1. explicit statement "import pyexcel.ext.xls" becomes a useless statement in your code. static code analyser(flake8/pep8/pycharm) would flag it up.
2. you have to explicitly import it. Otherwise, your plugin is not imported. PR 7 of pyexcel-io has extended discussion on this topic.
3. flask extension management system become deprecated by itself in Flask's recent development since 2016.

In order to overcome those shortcomings, implicit imports were coded into module's *__init__.py*. By iterating through currently installed modules in your python environment, the relevant plugins are imported automatically.

lml uses implicit import. In order to manage the plugins, pip can be used to install cherry-picked plugins or to remove unwanted plugins. In the situation where two plugins perform the same thing but have to co-exist in your current python path, you can nominate one plugin to be picked.

#### Plugin registration

In terms of plugin registrations, three different approaches have been tried. Monkey-patching was easy to implement. When a plugin is imported, it loads the plugin dictionary from the main package and add itself. But it is generally

[1] https://github.com/pyexcel/pyexcel
[2] http://zopecomponent.readthedocs.io/en/latest/
[3] https://packaging.python.org/namespace_packages/
[4] https://bitbucket.org/birkenfeld/sphinx-contrib/

perceived as a "bad" idea. Another way of doing it is to place the plugin code in the main component and the plugin just need to declare a dictionary as the plugin's meta data. The main package register the meta data when it is imported. tablib[5] uses such a approach. The third way is to use meta-classes. M. Alchin (2008)[6] explained how meta class can be used to register plugin classes in a simpler way.

lml uses meta data for plugin registration. Since lml load your plugin later, the meta data is stored in the module's __init__.py. For example, to load plugins later in tablib, the 'exports' variable should be taken out from the actual class file and replace the hard reference to the classes with class path string.

#### Plugin distribution

yapsy[7] and GEdit plugin management system[8] load plugins from file system. To install a plugin in those systems, is to copy and paste the plugin code to a designated directory. zope components, namespace packages and flask extensions can be installed via pypi. lml support the latter approach. lml plugins can be released to pypi and be installed by your end developers.

#### Design principle

To use lml, it asks you to avoid importing your "heavy" dependencies in *__init__.py*. lml respects the independence of individual packages. You can put modular level functions in your __init__.py as long as it does not trigger immediate import of your dependency. This is to allow the individual plugin to become useful as it is, rather to be integrated with your main package. For example, pyexcel-xls can be an independent package to read and write xls data, without pyexcel.

With lml, as long as your third party developer respect the plugin name prefix, they could publish their plugins as they do to any normal pypi packages. And the end developer of yours would only need to do pip install.

#### References

### 2.1.2 Tutorial

In this tutorial, we are going to go through various ways to build the command line application: **Robot Chef**. One is to build it as a single package. Another is to build it using lml: one main component with many plugins which are separately installable. By comparing the different approaches to build Robot Chef, we could see how lml can be used in practice.

**Robot Chef** would report what it knows about the food in the world. For example:

```
$ robotchef "Portable Battery"
I can cook Portable Battery for robots
```

When you type "Fish and Chips", it could reports it does not know:

```
$ robotchef "Fish and Chips"
I do not know how to cook Fish and Chips
```

For it to understand all the cuisines in the world, there are two ways to enlarge its knowledge base: one is obviously to grow by itself. the other is to open the api interface so that others could join your effort.

[5] https://github.com/kennethreitz/tablib

[6] M. Alchin, 2008, A Simple Plugin Framework, http://martyalchin.com/2008/jan/10/simple-plugin-framework/

[7] http://yapsy.sourceforge.net/

[8] https://wiki.gnome.org/Apps/Gedit/PythonPluginHowToOld

### Robot Chef all in one package without lml

In this chapter, we are going to see how **Robot Chef** could be implemented without lml. In later on chapters, we will bring in **lml** step by step.

### Demo

Please checkout the robot chef example:

```
$ git clone https://github.com/python-lml/robotchef_allinone
$ cd robotchef_allinone
$ python setup.py install
```

And then you could try:

```
$ robotchef_allinone "Fish and Chips"
I can fry Fish and Chips
```

### Conventional plugin and plugin factory

### plugin.py

*Chef* is the plugin interface that makes food. Boost, Bake and Fry are the actual implementations. Boost are for "robots". Bake and Fry are for human.

**Note:** The plugin interface is your responsibility. **lml** gives the freedom to you.

```
class Chef(object):
    def make(self, **params):
        print("I am a chef")


class Boost(Chef):
    def make(self, food=None, **keywords):
        print("I can cook %s for robots" % food)


class Fry(Chef):
    def make(self, food=None):
        print("I can fry " + food)


class Bake(Chef):
    def make(self, food=None):
        print("I can bake " + food)


PLUGINS = {
    "Portable Battery": Boost,
    "Fish and Chips": Fry,
```

(continues on next page)

(continued from previous page)

```
    "Cornish Scone": Bake,
    "Jacket Potato": Bake,
}
```

Line 13, class *Chef* defines the plugin class interface. For robotchef, *make* is defined to illustrate the functionality. Naturally you will be deciding the interface for your plugins.

Some of you might suggest that class *Chef* is unnecessary because Python uses duck-typing, meaning as long as the plugin has *make* method, it should work. Yes, it would work but it is a short term solution. Look at the long term, you could pass on additional functionalities through class *Chef* without touching the plugins. What's more, for plugin developers, a clear defined interface is better than no class at all. And I believe the functions of a real plugin are more than just one here.

Next in the plugin.py file, PLUGINS is the dictionary that has food name as key and Chef descendants as values. *get_a_plugin* method returns a Chef or raises NoChefException.

```
def get_a_plugin(food_name=None, **keywords):
    plugin = PLUGINS.get(food_name)
    if plugin is None:
        raise NoChefException("Cannot find a chef")
    plugin_cls = plugin()
    return plugin_cls
```

### main.py

Let us glimpse through the main code:

```
import sys

import robotchef_allinone.plugin as cuisine_manager


def main():
    if len(sys.argv) < 2:
        sys.exit(-1)

    food_name = sys.argv[1]
    try:
        knowledged_chef = cuisine_manager.get_a_plugin(food_name)
        knowledged_chef.make(food=food_name)
    except cuisine_manager.NoChefException:
        print("I do not know how to cook " + food_name)
```

The code takes the first command option as food name and feeds it to the factory method *get_a_plugin*, which returns a Chef to "make" the food. If no chef was found, it prints the default string: I do not know.

That is all about the all in one **Robot Chef**.

### Robot Chef all in one package with lml

Now let us bring in lml and see see how the lml package can be used to rewrite **Robot Chef** but in a single package. This chapter introduces two classes: *lml.plugin.PluginManager* and *lml.plugin.PluginInfo*. And show how those classes can be used to make factory pattern. Meanwhile, it demonstrates that the lml based plugins can be made to load immediately and in a single package. And this sections helps you to understand the next section where we will make the plugins to be loaded later.

### Demo

Please navigate to robotchef_allinone_lml and its packages. Do the following:

```
$ git clone https://github.com/python-lml/robotchef_allinone_lml
$ cd robotchef_allinone_lml
$ python setup.py install
```

And then you could try:

```
$ robotchef_allinone_lml "Fish and Chips"
I can fry Fish and Chips
```

### Lml plugins and plugin manager

### plugin.py

*CuisineManager* inherits from *PluginManager* class and replaces the static registry *PLUGINS* and the modular function *get_a_plugin*. Please note that *CuisineManager* declares that it is a manager for plugin_type named **cuisine**.

```
class CuisineManager(PluginManager):
    def __init__(self):
        PluginManager.__init__(self, "cuisine")

    def get_a_plugin(self, food_name=None, **keywords):
        return PluginManager.get_a_plugin(self, key=food_name, **keywords)

    def raise_exception(self, key):
        raise NoChefException("Cannot find a chef")
```

Next, the *PluginInfo* decorates all Chef's subclasses as **cuisine** plugins and register the decorated classes with the manager class for **cuisine**, *CuisineManager*. The food names become the tags which will be used to look up the classes.

```
from lml.plugin import PluginInfo, PluginManager

class Chef(object):
    def make(self, **params):
        print("I am a chef")


@PluginInfo("cuisine", tags=["Portable Battery"])
class Boost(Chef):
    def make(self, food=None, **keywords):
        print("I can cook %s for robots" % food)


@PluginInfo("cuisine", tags=["Fish and Chips"])
class Fry(Chef):
    def make(self, food=None):
        print("I can fry " + food)
```

(continues on next page)

(continued from previous page)

```
@PluginInfo("cuisine", tags=["Cornish Scone", "Jacket Potato"])
class Bake(Chef):
    def make(self, food=None):
        print("I can bake " + food)
```

Here is the rb3-diff-rb0-plugin.

### main.py

The main code has been updated to reflect the changes in plugin.py. *CuisineManager* has to be instantiated to be the a factory manager.

```
--- /home/docs/checkouts/readthedocs.org/user_builds/lml/checkouts/stable/examples/
↪robotchef_allinone/robotchef_allinone/main.py
+++ /home/docs/checkouts/readthedocs.org/user_builds/lml/checkouts/stable/examples/
↪robotchef_allinone_lml/robotchef_allinone_lml/main.py
@@ -1,15 +1,17 @@
 import sys

-import robotchef_allinone.plugin as cuisine_manager
+from robotchef_allinone_lml.plugin import CuisineManager, NoChefException


 def main():
     if len(sys.argv) < 2:
         sys.exit(-1)

+    cuisine_manager = CuisineManager()
+
     food_name = sys.argv[1]
     try:
         knowledged_chef = cuisine_manager.get_a_plugin(food_name)
         knowledged_chef.make(food=food_name)
-    except cuisine_manager.NoChefException:
+    except NoChefException:
         print("I do not know how to cook " + food_name)
```

Remember this interaction:

```
$ robotchef "Portable Battery"
I can cook Portable Battery for robots
```

The response comes from class *Boost*. It is obtained via CuisineManager when user types 'Portable Battery'. And the food parameter was passed to the instance of Boost. *make* method was called and it prints 'I can cook Portable Battery for robots'.

### See also

1. pyexcel-chart: use lml to refactor existing plugins

### Robot Chef distributed in multiple packages

In previous chapter, **Robot Chef** was written using lml but in a single package and its plugins are loaded immediately. In this chapter, we will decouple the plugin and the main package using lml. And we will demonstrates the changes needed to plugin them back with the main package.

### Demo

Do the following:

```
$ git clone https://github.com/python-lml/robotchef
$ cd robotchef
$ python setup.py install
```

The main command line interface module does simply this:

```
$ robotchef "Portable Battery"
I can cook Portable Battery for robots
```

Although it does not understand all the cuisines in the world as you see as below:

```
$ robotchef "Jacket Potato"
I do not know how to cook Jacket Potato
```

it starts to understand it once you install Chinese cuisine package to complement its knowledge:

```
$ git clone https://github.com/python-lml/robotchef_britishcuisine
$ cd robotchef_britishcuisine
$ python setup.py install
```

And then type in the following:

```
$ robotchef "Fish and Chips"
I can fry Fish and Chips
```

The more cuisine packages you install, the more dishes it understands. Here is the loading sequence:

### Decoupling the plugins with the main package

In order to demonstrate the capabilities of lml, *Boost* class is singled out and placed into an internal module **robotchef.robot_cuisine**. *Fry* and *Bake* are relocated to **robotchef_britishcuisine** package, which is separately installable. built-in and standalone-plugin will explain how to *glue* them up.

After the separation, in order to piece all together, a special function `lml.loader.scan_plugins()` needs to be called before the plugins are used.

```
--- /home/docs/checkouts/readthedocs.org/user_builds/lml/checkouts/stable/examples/
↪robotchef_allinone_lml/robotchef_allinone_lml/main.py
+++ /home/docs/checkouts/readthedocs.org/user_builds/lml/checkouts/stable/examples/
↪robotchef/robotchef/main.py
@@ -1,6 +1,16 @@
```

(continues on next page)

(continued from previous page)

```
 import sys
+import logging
+import logging.config

-from robotchef_allinone_lml.plugin import CuisineManager, NoChefException
+from lml.loader import scan_plugins_regex
+from robotchef.plugin import CuisineManager, NoChefException
+
+logging.basicConfig(
+    format="%(name)s:%(lineno)d - %(levelname)s - %(message)s",
+    level=logging.DEBUG,
+)
+
+BUILTINS = ["robotchef.robot_cuisine"]


 def main():
@@ -8,6 +18,11 @@
         sys.exit(-1)

     cuisine_manager = CuisineManager()
+    scan_plugins_regex(
+        plugin_name_patterns="robotchef_",
+        pyinstaller_path="robotchef",
+        white_list=BUILTINS,
+    )

     food_name = sys.argv[1]
     try:
```

What's more, `lml.loader.scan_plugins()` search through all installed python modules and register plugin modules that has prefix "**robotchef_**".

The second parameter of scan_plugins is to inform pyinstaller about the package path if your package is to be packaged up using pyinstaller. *white_list* lists the built-ins packages.

Once scan_plugins is executed, all 'cuisine' plugins in your python path, including the built-in ones will be discovered and will be collected by *PluginInfoChain* in a dictionary for `get_a_plugin()` to look up.

### Plugin management

As you see in the class relationship diagram, There has not been any changes for *CuisineManager* which inherits from *:class:lml.PluginManager* and manages **cuisine** plugins. Please read the discussion in *previous chapter*. Let us look at the plugins.

### Built-in plugin

*Boost* plugin has been placed in a submodule, **robotchef.robot_cuisine**. Let us see how it was done. The *magic* lies in robot_cuisine module's __init__.py

```
from lml.plugin import PluginInfoChain

PluginInfoChain(__name__).add_a_plugin(
```

(continues on next page)

(continued from previous page)

```
    "cuisine", "electrify.Boost", tags=["Portable Battery"]
)
```

A unnamed instance of *lml.plugin.PluginInfoChain* registers the meta data internally with *CuisineManager*. *__name__* variable refers to the module name, and in this case it equals 'robotchef.robot_cuisine'. It is used to form the absolute import path for *Boost* class.

First parameter **cuisine** indicates that *electrify.Boost* is a **cuisine** plugin. **lml** will associate it with *CuisineManager*. It is why CuisineMananger has initialized as 'cuisine'. The second parameter is used the absolute import path 'robotchef.robot_cuisine.electricity.Boost'. The third parameter *tags* are the dictionary keys to look up class *Boost*.

Here is a warning: to achieve lazy loading as promised by **lml**, you shall avoid heavy duty loading in __init__.py. this design principle: **not to import any un-necessary modules in your plugin module's __init__.py**.

That's all you need to write a built-in plugin.

### Standalone plugin

Before we go to examine the source code of **robotchef_britishcuisine**, please let me dictate that the standalone plugins shall respect the package prefix, which is set by the main package. In this case, the plugin packages shall start with '**robotchef_**'. Hence for British Cuisine, it is named as 'robotchef_britishcuisine'.

Now let us have look at the module's __init__.py, you would find similar the plugin declaration code as in the following. But nothing else.

```
from lml.plugin import PluginInfoChain

PluginInfoChain(__name__).add_a_plugin(
    "cuisine", "fry.Fry", tags=["Fish and Chips"]
).add_a_plugin("cuisine", "bake.Bake", tags=["Cornish Scone", "Jacket Potato"])
```

Because we have relocated *Fry* and *Bake* in this package, the instance of *PluginInfoChain* issues two chained call `add_a_plugin()` but with corresponding parameters.

---

**Note:** In your plugin package, you can add as many plugin class as you need. And the tags can be as long as you deem necessary.

---

Let me wrap up this section. All you will need to do, in order to make a standalone plugin, is to provide a package installer(setup.py and other related package files) for a built-in plugin.

### The end

That is all you need to make your main component to start using component based approach to expand its functionalities. Here is the takeaway for you:

1. *lml.plugin.PluginManager* is just another factory pattern that hides the complexity away.
2. You will need to call `lml.loader.scan_plugins()` in your __init__.py or where appropriate before your factory class is called.

### More standalone plugins

You are left to install robotchef_chinesecuisine and robotchef_cook yourself and explore their functionalities.

### How to ask robotchef to forget British cuisine?

The management of standalone plugins are left in the hands of the user. To prevent robotchef from finding British cuisine, you can use pip to uninstall it, like this:

```
$ pip uninstall robotchef_britishcuisine
```

### Robot Chef version 2: Use lml to write a shared library

In previous chapter, lml was used to split all in one Robot Chef into one core package and several plugins module and packages. In this chapter, we are going to go one step further to split the core package into two so as to showcase how to use lml to write a shared api library.

### Demo

Please checkout the following examples:

```
$ virtualenv --no-site-packages robotchefv2
$ source robotchefv2/bin/activate
$ git clone https://github.com/python-lml/robotchef_v2
$ cd robotchef_v2
$ python setup.py install
$ cd ..
$ git clone https://github.com/python-lml/robotchef_api
$ cd robotchef_api
$ python setup.py install
```

And then you can type in and test the second version of Robot Chef:

```
$ robotchef_v2 "Portable Battery"
I can cook Portable Battery for robots
$ robotchef_v2 "Jacket Potato"
I do not know how to cook Jacket Potato
```

In order to add "Jacket Potato" in the know-how, you would need to install robotchef_britishcuisine in this folder:

```
$ git clone https://github.com/python-lml/robotchef_britishcuisine_v2
$ cd robotchef_britishcuisine_v2
$ python setup.py install
$ robotchef_v2 "Jacket Potato"
I can bake Jacket Potato
```

### Robot Chef v2 code

Let us look at main code robotchef_v2:

```
--- /home/docs/checkouts/readthedocs.org/user_builds/lml/checkouts/stable/examples/
↪robotchef/robotchef/main.py
+++ /home/docs/checkouts/readthedocs.org/user_builds/lml/checkouts/stable/examples/v2/
↪robotchef_v2/robotchef_v2/main.py
@@ -1,28 +1,11 @@
```

(continues on next page)

(continued from previous page)

```
 import sys
-import logging
-import logging.config

-from lml.loader import scan_plugins_regex
-from robotchef.plugin import CuisineManager, NoChefException
-
-logging.basicConfig(
-    format="%(name)s:%(lineno)d - %(levelname)s - %(message)s",
-    level=logging.DEBUG,
-)
-
-BUILTINS = ["robotchef.robot_cuisine"]
+from robotchef_api import NoChefException, cuisine_manager


 def main():
     if len(sys.argv) < 2:
         sys.exit(-1)
-
-    cuisine_manager = CuisineManager()
-    scan_plugins_regex(
-        plugin_name_patterns="robotchef_",
-        pyinstaller_path="robotchef",
-        white_list=BUILTINS,
-    )

     food_name = sys.argv[1]
     try:
```

The code highlighted in red are removed from main.py and are placed into **robotchef_api** package. And robotchef_v2 becomes the consumer of the robotchef api.

And plugin.py and robot_cuisine has been moved to **robotchef_api** package.

### Robot Chef API

Now let us look at robotchef_api. In the following directory listing, the plugin.py And robot_cuisine is exactly the same as the plugin.py and robot_cuisine in robotchef:

```
__init__.py    plugin.py       robot_cuisine
```

Notably, the plugin loader is put in the __init__.py:

```
from lml.loader import scan_plugins_regex
from robotchef_api.plugin import CuisineManager, NoChefException  # noqa: F401

BUILTINS = ["robotchef_api.robot_cuisine"]


scan_plugins_regex(
    plugin_name_patterns="^robotchef_.*$",
    pyinstaller_path=__path__,  # noqa: F821
    white_list=BUILTINS,
)
cuisine_manager = CuisineManager()
```

scan_plugins_regex here loads all modules that start with "**robotchef_**" and as well as the module *robotchef_api.robot_cuisine* in the white_list.

This is how you will write the main component as a library.

#### Built-in plugin and Standalone plugin

You may have noticed that a copy of robotchef_britishcuisine is placed in v2 directory. Why not using the same one above v2 directory? although they are almost identical, there is a minor difference. robotchef_britishcuisine in v2 directory depends on robotchef_api but the other British cuisine package depends on robotchef. Hence, if you look at the fry.py in v2 directory, you will notice a slight difference:

```
--- /home/docs/checkouts/readthedocs.org/user_builds/lml/checkouts/stable/examples/
↪robotchef_britishcuisine/robotchef_britishcuisine/fry.py
+++ /home/docs/checkouts/readthedocs.org/user_builds/lml/checkouts/stable/examples/v2/
↪robotchef_britishcuisine/robotchef_britishcuisine/fry.py
@@ -1,4 +1,4 @@
-from robotchef.plugin import Chef
+from robotchef_api.plugin import Chef


 class Fry(Chef):
```

#### Additional references

1. pyexcel-chart: use lml to refactor existing plugins

### 2.1.3 Logging facility

During the development of lml package, the logging facility helps debugging a lot. Let me show you how to enable the logs of lml.

#### Enable the logging

Let us open robotchef's main.py. Insert the highlighted codes.

```
import sys

from robotchef.plugin import CuisineManager, NoChefException

import logging
import logging.config

logging.basicConfig(
    format='%(name)s:%(lineno)d - %(levelname)s - %(message)s',
    level=logging.DEBUG)


def main():
    if len(sys.argv) < 2:
        sys.exit(-1)
```

(continues on next page)

(continued from previous page)

```
    manager = CuisineManager()
...
```

Then you will need to run the installation again:

```
$ cd robotchef
$ python setup.py install
```

Let us run the command again:

```
$ robotchef "Jacket Potato"
lml.plugin:226 - DEBUG - declare 'cuisine' plugin manager
lml.loader:52 - DEBUG - scanning for plugins...
lml.utils:48 - DEBUG - found robotchef_allinone_lml
lml.plugin.PluginInfoChain:139 - DEBUG - add robotchef_britishcuisine.fry.Fry as
↪'cuisine' plugin
robotchef.plugin.CuisineManager:178 - DEBUG - load robotchef_britishcuisine.fry.Fry␣
↪later
lml.plugin.PluginInfoChain:139 - DEBUG - add robotchef_britishcuisine.bake.Bake as
↪'cuisine' plugin
robotchef.plugin.CuisineManager:178 - DEBUG - load robotchef_britishcuisine.bake.Bake␣
↪later
lml.utils:48 - DEBUG - found robotchef_britishcuisine
lml.plugin.PluginInfoChain:139 - DEBUG - add robotchef.robot_cuisine.electrify.Boost␣
↪as 'cuisine' plugin
robotchef.plugin.CuisineManager:178 - DEBUG - load robotchef.robot_cuisine.electrify.
↪Boost later
lml.utils:48 - DEBUG - found robotchef.robot_cuisine
lml.loader:82 - DEBUG - scanning done
robotchef.plugin.CuisineManager:160 - DEBUG - get a plugin called
robotchef.plugin.CuisineManager:210 - DEBUG - import robotchef_britishcuisine.bake.
↪Bake
robotchef.plugin.CuisineManager:202 - DEBUG - load <class 'robotchef_britishcuisine.
↪bake.Bake'> now for 'Jacket Potato'
I can bake Jacket Potato
```

Reading the log with the loading sequence,

Three Chef plugins were discovered: robotchef_britishcuisine.fry.Fry, robotchef_britishcuisine.bake.Bake and robotchef.robot_cuisine.electricity.Boost. However, they are not imported yet. When the robotchef try to look up a plugin, it logs "get a plugin called". And it is actual time when a plugin is imported.

### 2.1.4 API documentation

#### lml.loader

Plugin discovery module. It supports plugins installed via pip tools and pyinstaller. `scan_plugins()` is expected to be called in the main package of yours at an earliest time of convenience.

**copyright**

(c) 2017-2020 by Onni Software Ltd.

**license** New BSD License, see LICENSE for more details

lml.loader.**scan_plugins_regex**(*plugin_name_patterns=None*, *pyinstaller_path=None*, *black_list=None*, *white_list=None*)

Implicitly discover plugins via pkgutil and pyinstaller path using regular expression

**plugin_name_patterns: python regular expression** it is used to match all your plugins, either it is a prefix, a suffix, some text in the middle or all.

**pyinstaller_path:string** used in pyinstaller only. When your end developer would package your main library and its plugins using pyinstaller, this path helps pyinstaller to find the plugins.

**black_list:list** a list of module names that should be skipped.

**white_list:list** a list of modules that comes with your main module. If you have a built-in module, the module name should be inserted into the list.

For example, robot_cuisine is a built-in module inside robotchef. It is listed in white_list.

### lml.plugin

lml divides the plugins into two category: load-me-later plugins and load-me-now ones. load-me-later plugins refer to the plugins were loaded when needed due its bulky and/or memory hungry dependencies. Those plugins has to use lml and respect lml's design principle.

load-me-now plugins refer to the plugins are immediately imported. All conventional Python classes are by default immediately imported.

*PluginManager* should be inherited to form new plugin manager class. If you have more than one plugins in your architecture, it is advisable to have one class per plugin type.

*PluginInfoChain* helps the plugin module to declare the available plugins in the module.

*PluginInfo* can be subclassed to describe your plugin. Its method `tags()` can be overridden to help its matching *PluginManager* to look itself up.

**copyright**

(c) 2017-2020 by Onni Software Ltd.

**license** New BSD License, see LICENSE for more details

**class** lml.plugin.**PluginInfo**(*plugin_type*, *abs_class_path=None*, *tags=None*, ***keywords*)

Information about the plugin.

It is used together with PluginInfoChain to describe the plugins. Meanwhile, it is a class decorator and can be used to register a plugin immediately for use, in other words, the PluginInfo decorated plugin class is not loaded later.

**name:** plugin name

**absolute_import_path:** absolute import path from your plugin name space for your plugin class

**tags:** a list of keywords help the plugin manager to retrieve your plugin

**keywords:** Another custom properties.

For load-me-later plugins:

```
>>> info = PluginInfo("sample",
...       abs_class_path='lml.plugin.PluginInfo', # demonstration only.
...       tags=['load-me-later'],
...       custom_property = 'I am a custom property')
>>> print(info.module_name)
lml
```

(continues on next page)

(continued from previous page)

```
>>> print(info.custom_property)
I am a custom property
```

For load-me-now plugins:

```
>>> @PluginInfo("sample", tags=['load-me-now'])
... class TestPlugin:
...     def echo(self, words):
...         print("echoing %s" % words)
```

Now let's retrive the second plugin back:

```
>>> class SamplePluginManager(PluginManager):
...     def __init__(self):
...         PluginManager.__init__(self, "sample")
>>> sample_manager = SamplePluginManager()
>>> test_plugin=sample_manager.get_a_plugin("load-me-now")
>>> test_plugin.echo("hey..")
echoing hey..
```

**class** lml.plugin.**PluginInfoChain**(*path*)

Pandas style, chained list declaration

It is used in the plugin packages to list all plugin classes

**class** lml.plugin.**PluginManager**(*plugin_type*)

Load plugin info into in-memory dictionary for later import

**plugin_type:** the plugin type. All plugins of this plugin type will be registered to it.

**dynamic_load_library**(*a_plugin_info*)

Dynamically load the plugin info if not loaded

**a_plugin_info:** a instance of plugin info

**get_a_plugin**(*key*, ***keywords*)

Get a plugin

**key:** the key to find the plugins

**keywords:** additional parameters for help the retrieval of the plugins

**load_me_later**(*plugin_info*)

Register a plugin info for later loading

**plugin_info:** a instance of plugin info

**load_me_now**(*key*, *library=None*, ***keywords*)

Import a plugin from plugin registry

**key:** the key to find the plugin

**library:** to use a specific plugin module

**raise_exception**(*key*)

Raise plugin not found exception

Override this method to raise custom exception

**key:** the key to find the plugin

**register_a_plugin**(*plugin_cls*, *plugin_info*)

for dynamically loaded plugin during runtime

**plugin_cls:** the actual plugin class refered to by the second parameter

**plugin_info:** a instance of plugin info

Beyond the documentation above, here is a list of projects using lml:

1. pyexcel
2. pyecharts
3. moban

lml is available on these distributions:

1. ARCH linux
2. Conda forge
3. OpenSuse

# Python Module Index

## l

# Index