apyshell

Release 1.0

Mark Anacker

Nov 21, 2023

CONTENTS

CHAPTER

ONE

APYSHELL PACKAGE

1.1 Subpackages

1.2 Submodules

1.2.1 apyshell.apyshell module

apyshell - Python Embedded apy script runnner

This module creates a framework for running lightweight scripts under the apyengine interpreter. It demonstrates embedding and controlling the engine, and extending the funcionality available to scripts.

It can be run either stand-alone, or itself embedded into an application.

Credits

- version: 1.0
- last update: 2023-Nov-21
- License: MIT
- Author: Mark Anacker <closecrowd@pm.me>
- Copyright (c) 2023 by Mark Anacker

apyshell(*script*, *basedir='/opt/apyshell/scripts'*, *extensiondir='/opt/apyshell/extensions'*, *extension_opts=None*, *args=None*, *initscript=None*, *globals=False*)
Execute a script file under anyengine.

This is the main entry point.

Args: script : The .apy script to execute.

Returns: The return value. True for success, False otherwise.

savepid(*pfile*)
Save our current PID if we can.

Note: this doesn't do any checking of the file path. Take care if running under root.

---

usage()
Help message.

1.2.2 apyshell.extensionapi module

extensionapi - The interface between extensions and the engine.

An instance of this class is passed to the extensions at load time to give them an API back into the engine via the ExtensionMgr

Credits

- version: 1.0
- last update: 2023-Nov-20
- License: MIT
- Author: Mark Anacker <closecrowd@pm.me>
- Copyright (c) 2023 by Mark Anacker

class ExtensionAPI(*eng*, *parent*, *internal=True*)

Bases: `object`

getSysvar_(*name*, *default=None*)
Return the value of a system variable or a default

getvar_(*vname*, *default=None*)
Return the value of a script variable or a default

handleEvents(*name*, *data*)
Handle callback events.

If an extension allows for handlers to receive events, each callback comes through here. This function passes the name and data object through to the ExtensionMgr to run in the engine.

Args:

name : The name of the handler func().

data : The object to pass as the **only** argument to the handler.

Returns:

ret : A string with the return from the handler()

None : Something failed

isDef_(*name*)
Return True is the name is defined

listDefs_(*exception_list=None*)
Return a list of script def procs

list_Modules_()
Return a list of install_()-ed modules

loadScript_(*filename*, *persist=False*)
Load and execute a script file

logError(*modname=''*, **args*)

regcmd(*name*, *func=None*)
Add a new command callable by scripts

registerCmds(*mdict*)
Make an extension's functions available to scripts

Extensions call here through the api reference to install their commands into the engine.

Args:

mdict : A dict with the commands and function refs.

Returns

True [Commands are installed and the extension is] ready to use.

False [Commands are NOT installed, and the extension] is inactive.

setSysvar_(*name*, *val*)
Set a system variable

setvar_(*vname*, *val*)
Set a script variable

unregisterCmds(*mdict*)
Remove an extension's commands from the engine.

Extensions call here through the api reference to remove their commands from the engine.

Args:

mdict : A dict with the commands and function refs.

Returns

True : Commands have been removed.

False : Something failed.

debug(**args*)

1.2.3 apyshell.extensionmgr module

extensionmgr - Handles extension load/unload for apyshell.

This module supports the extension handling commands for apyshell.

Credits

- version: 1.0
- last update: 2023-Nov-20
- License: MIT
- Author: Mark Anacker <closecrowd@pm.me>
- Copyright (c) 2023 by Mark Anacker

class ExtensionMgr(*eng*, *epath*, *options=None*)
Bases: `object`

__init__(*eng*, *epath*, *options=None*)
Constructs an instance of the ExtensionMgr class.

This instance manages the loading and unloading of extension modules in an apyengine instance. Extensions add new functions callable by scripts.

Args:

eng : The instance of ApyEngine to manage.

epath : The path or list of paths to allowed extensions.

options : A dict of extension options passed down to all extensions.

Returns:

Nothing

handleEvents(*name*, *data*)
Handle callback events.

If an extension allows for handlers to receive events, each callback comes through here. This function makes sure the handler (a def func() in the script) is defined. It then executes the handler func **in the context of the thread that calls this method.** This may not be the thread that the majority of the script is running on.

Args:

name : The name of the handler func().

data : The object to pass as the **only** argument to the handler.

Returns:

ret : A string with the return from the handler()

None : Something failed

isExtLoaded_(*ename*)
Handles the isExtLoaded_() function.

This function is used to test whether an extension is currently loaded or not.

Args:

ename : The name of the extension to check.

Returns:

True : The extension is loaded.

False : It's not loaded.

listExtensions_()
Handles the listExtensions_() function.

Return a list[] of currently-loaded extensions - those that a script has used loadExtension_() to install.

Args:

None

Returns:

A list[] of the loaded extension names.

None if there are no loaded extensions.

loadExtension_(*ename*)
Handles the loadExtension_() function.

This function loads an extension by name (no paths allowed), if it's in the __availExtensions list. If the list is empty, the extensions directories are scanned.

If the extension is available, it's loaded into the engine's symbol table and the register() function is called to add it's functions.

Args:

ename : The name of the extension to load.

Returns:

True : The extension loaded correctly.

False : It didn't load.

register()

Make this extension's functions available to scripts

This method installs our script API methods as functions in the engine symbol table, making them available to scripts.

This is called by the host application right after the ExtensionMgr object is instanciated.

Example:

```
# create the extension manager

emgr = ExtensionMgr(engine, extensiondir, extension_opts)

# register it's commands

emgr.register()
```

Note:

Functions installed:

- scanExtensions_() : Look through the extensions dir and return a list of available extensions.
- listExtensions_() : Return a list of loaded extensions.
- isExtLoaded_() : Return True if the extension is loaded.
- loadExtension_() : Load an extension into the engine.
- unloadExtension_() : Remove an extension from the engine.

Args:

None.

Returns

Nothing.

---

scanExtensions_()

Handles the scanExtensions_() function.

This function builds a list of the Python files in the extension director(ies) stored in self.__expath. This can be either a single directory, or a list of them. All .py files in these directories are presumed to be extensions, although there is a signature test performed at extension load time.

You can limit which extensions are available simply by limiting which files are placed into these directories.

Args:

None

Returns:

A list[] of the available extension names.

None if there are no extensions.

scanForExtensions(*dir*)
Scan the extension dirs for modules.

This method wlaks through the supplied entension paths and records all of the Python modules. This creates the list of available extensions.

shutdown()
Perform a graceful shutdown.

Calls the shutdown() method of each loaded extension, making sure they go cleanly.

This gets called by apyshell just before is exits.

unloadExtension_(*ename*)
Handles the unloadExtension_() function.

This function removes a currently-loaded extension from the symbol table, making it unavailable to scripts. It calls the shutdown() and unregister() methods in the extension before removing it.

Args:

ename : The name of the extension to unload.

Returns:

True : The extension unloaded correctly.

False : There was an error.

quoteSpecial(*orig*)

1.2.4 apyshell.support module

support - Various stand-alone support functions for apyshell.

This file contains various utility functions used by apyshell.

Credits

- version: 1.0
- last update: 2023-Nov-21
- License: MIT
- Author: Mark Anacker <closecrowd@pm.me>
- Copyright (c) 2023 by Mark Anacker

checkFileName(*fname*)
Check a file name

Checks the given filename for characters in the set of a-z, A-Z, _ or -

Args: fname : The name to check

Returns: True if the name is entirely in that set False if there were invalid char(s)

debugMsg(*source=''*, **args*)

enableDebug(*arg*)

errorMsg(*source=''*, **args*)

get_queue(*q*, *defvalue=None*, *timeout=0*)
Get the next value from a Queue.

This function does a blocking GET from a Queue, returning the next item. If the timeout is > 0 and expires, or there was an error on the get(), return a default value.

getparam(*table*, *key*, *default*, *remove=False*)
Destructively return a dict entry.

Return a value from a dict, optionally removing it from the dict after grabbing it. This is mainly used to remove options from a **kwargs parameter before passing it down to a lower level function.

If the key is not found in the dict, return a default value. Remove has no effect in this case.

Args:

table : The dict object to modify.

key : The key to look up.

default : A default value to return if key isn't found.

remove : If True, remove the entry if found.

Returns:

The value from the dict, or the default.

retError(*api*, *module*, *msg*, *ret=False*)
Logging error return.

Extensions call this function to log an error message through the ExtensionMgr API, then pass up an error value.

Args:

api : The extensionmgr api reference.

module : A string with the failed module name.

msg : The actual error message.

ret : The value to return.

Returns:

The value of ret, or False if it's missing.

sanitizePath(*path*)
Clean a path.

Remove dangerous characters from a path string.

Args:

path : The string with the path to clean

Returns:

The cleaned path or None if there was a problem

setparam(*table*, *key*, *value*, *replace=False*)
Add or update a value in a dict.

This function adds a value to a dict, or optionally replaces an existing value. This is mainly used to manipulate values in **kwargs parameters.

If the key is not found in the table, add it and it's value. Replace has no effect in this case.

Args:

table : The dict object to modify.

key : The key to look up.

value : The value to add or replace

replace : If True, update an existing value if found.

Returns:

Nothing.

swapparam(*table*, *oldkey*, *newkey*, *value=None*)
Move a dict entry to a new key.

This function will move a dict entry from an old key to a new key, optionally replacing the value at the same time. If the old key doesn't exist, a new entry is created.

unlock__(*lock*)
Unlock a locked mutex.

Utility function to unlock a mutex if it's currently locked.

Args:

lock : An instance of threading.Lock().

Returns:

Nothing.

1.3 Module contents

ApyShell - Python Embedded apy script runnner

This package creates a framework for running lightweight scripts under the apyengine interpreter. It demonstrates embedding and controlling the engine. It can be run either stand-alone, or itself embedded into an application.

The companion project "apyengine" has documentation and examples of how to use the apyengine outside of this package. The entire apyengine package is included in this package for convenience. <https://github.com/closecrowd/apyengine>

Credits

- version: 1.0
- last update: 2023-Nov-17
- License: MIT
- Author: Mark Anacker <closecrowd@pm.me>
- Copyright (c) 2023 by Mark Anacker

PYTHON MODULE INDEX

a

INDEX

Symbols

A

C

D

E

G

H

I

L

M

Q

R

S

U