Ərk Plugins

How to write plugins for the Ərk IRC Client

https://github.com/nutjob-laboratories/erk
https://github.com/nutjob-laboratories/erk-plugins

An Ərk plugin is a Python 3 class that inherits from **`erk.Plugin`**.

Summary

An Ərk ***plugin*** is a file containing one of more Python 3 classes that inherit from **`erk.Plugin`**. A ***package*** is a directory that contains one or more files containing Ərk plugins, along with other optional files that either contain metadata or graphics.

In this document, the term "plugin" is used for both plugins and packages; if a distinction between the two terms is necessary, the correct term is used.

Installing Ərk Plugins

Ərk plugins "live" in a directory named "plugins", in the main Ərk installation directory. To install a plugin, simply place it in the "plugins" directory. To install exported plugin packages created with the Ərk editor, click on the "Install plugin" entry in the "Plugins" menu and select the zip file containing the package; the zip file will be extracted into Ərk's plugin directory.

Uninstalling Ərk Plugins

Delete (or otherwise remove) the plugin file or package from the "plugins" directory. Alternately, click on the "Uninstall plugin" entry in the "Plugins" menu.

Writing an Ərk Plugin

The first step is creating a new Python 3 class that inherits from **`erk.Plugin`**. To get access to this class, import it from Ərk:

```
1 from erk import *
```

Then, create the rest of the class. The following plugin will do *nothing*, but it's a complete example that shows what a plugin should look like:

```
 1 from erk import *
 2
 3 class ExamplePlugin(Plugin):
 4      def __init__(self):
 5           self.name = "An Example Plugin"
 6           self.description = "Example plugin for documentation"
 7
 8      def received(self,client,line):
 9           pass
10
11      def sent(self,client,line):
12           pass
13
14      def load(self):
15           pass
16
17      def unload(self):
18           pass
19
20      def input(self,client,name,text):
21           pass
22
23      def connect(self,client):
24           pass
25
26      def public(self,client,channel,user,message):
27           pass
28
29      def private(self,client,user,message):
30           pass
31
32      def notice(self,client,target,user,message):
33           pass
34
35      def join(self,client,channel,user):
36           pass
37
38      def part(self,client,channel,user):
39           pass
40
41      def tick(self,client):
42           pass
43
44      def ctcp(self,client,user,channel,tag,message):
45           pass
```

Plugin Attributes

Plugins have only two *required* attributes: **`name`** and **`description`**. Plugins can also possess four other optional attributes: **`author`**, **`version`**, **`source`**, and **`website`**.

`name`

This is the name of the plugin. It must be unique (that is, no other loaded plugin can use the same name). It can contain spaces. All plugins are required to have a **`name`** attribute.

`description`

This describes the plugin. It can also contain spaces, but it does not have to be unique. All plugins are required to have a **`description`** attribute.

`author`

The name of the person (or persons) who created/maintain the plugin.

`version`

A string representing the plugin's version.

`source`

A URL to where users can obtain the source code or information about the source code of the plugin.

`website`

A URL where users can obtain the plugin, information about the plugin, or information about the person (or persons) who wrote the plugin.

Plugin Methods

There are five methods a plugin can have; they are not required to have any specific methods, but *they must have at least one of the methods* to be a valid plugin.

Most of the methods are passed an instance of **`twisted.words.protocols.irc.IRCClient`** as a first argument. This is the instance that Ərk uses to communicate with the IRC connection associated with the event or window that triggered the method's execution. At this time, Ərk uses version 19.2 of Twisted; the documentation for this object can be found at https://twistedmatrix.com/documents/current/api/twisted.words.protocols.irc.IRCClient.html . The instance has been modified so that any outgoing messages are displayed in the client (so, if you use the **`msg()`** method to send a PRIVMSG command, the message sent will be displayed in the appropriate channel or private chat window).

`load`

Arguments	None
Description	This method is executed as soon as the plugin is loaded into memory.

unload

Arguments None

Description This method is executed when the plugin is unloaded from memory; if not otherwise unloaded, this method is executed when Ərk shuts down.

input

Arguments **client** (instance of **twisted.words.protocols.irc.IRCClient**), **name** (string), **text** (string)

Description This method is executed whenever a user enters text into the Ərk IRC client and presses enter. **name** contains the "name" of the window the text was entered into. If entered into a non-chat server console, **name** will be set to "_Server"; other wise, **name** will contain the name of the channel the text was entered into (if entered into a channel chat window) or the nickname of a user (if entered into a private chat window). **text** contains the text that was entered into the window.

This method is intended to be used to implement new commands for the Ərk client. To prevent any further processing of user input, have this method return **True**. *Be careful with this*. It's trivially easy to implement a plugin that can prevent *any* user input at all:

```
1 def input(self,client,name,text):
2     return True
```

Only return **True** if no further processing of the text is required; for example, when implementing a custom command, returning **True** prevents the user's command input being sent to the server as a PRIVMSG.

Ərk will try to detect any plugins that maliciously prevent user input and refuse to load them; this is not perfect (or even close to perfect) and should not be relied on.

public

Arguments **client** (instance of **twisted.words.protocols.irc.IRCClient**), **channel** (string), **user** (string), **message** (string)

Description This method is executed when the Ərk client receives a public (channel) chat message. **channel** contains the name of the channel the chat originated from. **user** contains information about the user that sent the message; this is presented in the format provided by the server: **nickname**! **username**@**hostname**. **message** contains the contents of the message sent.

private

Arguments **client** (instance of **twisted.words.protocols.irc.IRCClient**), **user** (string), **message** (string)

Description This method is executed when the Ərk client receives a private chat message. **user** contains information about the user that sent the message; this is presented in the format provided by the server: **nickname**! **username**@**hostname**. **message** contains the contents of the message sent.

tick

Arguments **client** (instance of **twisted.words.protocols.irc.IRCClient**)

Description This method is executed every second the Ərk client is connected to a server.

join

Arguments **client** (instance of **twisted.words.protocols.irc.IRCClient**), **channel** (string), **user** (string)

Description This method is executed when the Ərk client receives a channel join notification. **channel** contains the name of the channel the user joined. **user** contains information about the user that joined; this is presented in the format provided by the server: **nickname**!**username**@**hostname**. **message** contains the contents of the message sent.

part

Arguments **client** (instance of **twisted.words.protocols.irc.IRCClient**), **channel** (string), **user** (string)

Description This method is executed when the Ərk client receives a channel part notification. **channel** contains the name of the channel the user left. **user** contains information about the user that left the channel; this is presented in the format provided by the server: **nickname**!**username**@**hostname**. **message** contains the contents of the message sent.

connect

Arguments **client** (instance of **twisted.words.protocols.irc.IRCClient**)

Description This method is executed when the Ərk client is completes registration with a server.

notice

Arguments **client** (instance of **twisted.words.protocols.irc.IRCClient**), **target** (string), **user** (string), **message** (string)

Description This method is executed when the Ərk client receives a notice message. **target** contains the name of the user or channel the notice was sent to. **user** contains information about the user that sent the message; this is presented in the format provided by the server: **nickname**! **username**@**hostname**. **message** contains the contents of the message sent.

ctcp

Arguments **client** (instance of **twisted.words.protocols.irc.IRCClient**), **user** (string), **channel** (string), **tag** (string), **message** (string)

Description This method is executed when the Ərk client receives an unrecognized CTCP message. **user** contains information about the user that sent the message; this is presented in the format provided by the server: **nickname**! **username**@**hostname**. **channel** contains the channel (or nickname) the message was sent from. **tag** contains the CTCP message tag sent in the message. **message** contains the contents of the message sent.

received

Arguments **client** (instance of **twisted.words.protocols.irc.IRCClient**), **line** (string)

Description This method is executed every time the Ərk client receives a line of data from the server.

sent

Arguments **client** (instance of **twisted.words.protocols.irc.IRCClient**), **line** (string)

Description This method is executed every time the Ərk client sends a line of data to the server.

Built-In Methods

Plugins also contain several built-in methods to interact with the Ərk IRC client.

`info`

Arguments	None
Returns	String
Description	Returns a string containing the application name and version of the Ərk IRC client.

`write`

Arguments	**`name`** (string), **`text`** (string)
Returns	Nothing
Description	Writes text to a specific Ərk chat window. **`write()`** can only write text to a window associated with the client that triggered the plugin method's execution. That means that **`write()`** can write to channel windows that the client has joined or private message sessions that are open and ongoing. **`write()`** cannot write text to windows associated with another client. **`name`** should contain the name of the channel or user nick whose window you want to write to; **`text`** should contain the text you want to write. **`text`** can contain HTML. Any text written will not be saved to the log, if logging is turned on. For example, to write to the chat window for the channel "#erk", you could use **`self.write("#erk","This is the text to write")`**. *You cannot **`write()`** text to windows associated with another client.* If you are trying to use **`write()`** from the **`input()`** method, for example, the only windows you can write to *must be on the same server the window that triggered the method is associated with*. This method will always fail in a plugin's **`load()`** and **`unload()`** methods.

`log`

Arguments	**`name`** (string), **`text`** (string)
Returns	Nothing
Description	Writes text to a specific Ərk chat window. **`log()`** can only write text to a window associated with the client that triggered the plugin method's execution. That means that **`log()`** can write to channel windows that the client has joined or private message sessions that are open and ongoing. **`log()`** cannot write text to windows associated with another client. **`name`**

should contain the name of the channel or user nick whose window you want to write to; **`text`** should contain the text you want to write. **`text`** can contain HTML. Any text written ***will*** be saved to the log, if logging is turned on.

For example, to write to the chat window for the channel "#erk", you could use **`self.log("#erk","This is the text to write")`**.

*You cannot **`log()`** text to windows associated with another client.* If you are trying to use **`log()`** from the **`input()`** method, for example, the only windows you can write to *must be on the same server the window that triggered the method is associated with*.

This method will always fail in a plugin's **`load()`** and **`unload()`** methods.

`console`

Arguments **`text`** (string)

Returns Nothing

Description Writes to the console window associated with the client that triggered the plugin method's execution. This method will always fail in a plugin's **`load()`** and **`unload()`** methods.

`print`

Arguments **`text`** (string)

Returns Nothing

Description Writes to the current window displayed in Ərk, whether it's a server console, channel chat, or private chat; if there is no current window, the method will fail silently and not display any text. This method will always fail in a plugin's **`load()`** and **`unload()`** methods.

`sysmsg`

Arguments **`text`** (string)

Returns Nothing

Description Writes a system message to the current window displayed in Ərk, whether it's a server console, channel chat, or private chat; if there is no current window, the method will fail silently and not display any text. This method will always fail in a plugin's **`load()`** and **`unload()`** methods.

exec

Arguments **`data`** (string)

Returns Returns **`True`** if the command was processed, and **`False`** if not

Description Processes a string as if it were input by a user into the current chat's text input. This allows plugins to execute commands without using the Twisted IRC client. If passed a string *without* a input command, the string will be passed to the server as chat. This method will always fail in a plugin's **`load()`** and **`unload()`** methods.

uptime

Arguments None

Returns Int

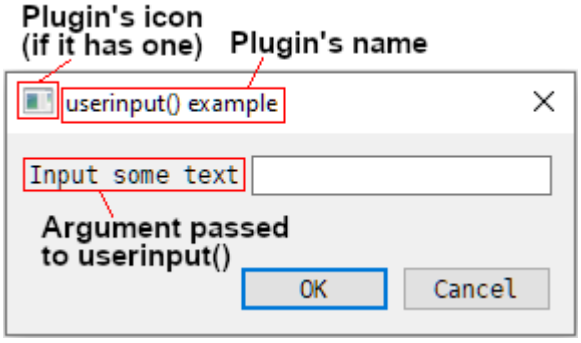
Description Returns the length of time (in seconds) that Ərk has been connected to the IRC server that triggered the plugin's execution. This method will always return 0 (zero) in a plugin's **`load()`** and **`unload()`** methods.

userinput

Arguments **`text`** (string)

Returns String if the user entered text into the dialog and clicked "Ok" or **`None`** if the user clicked "Cancel".

Description Opens a dialog to get information from the user; the dialog will contain the plugin's name as the title, and **`text`** will appear before the text entry widget.



`msgbox`

Arguments **`text`** (string)

Returns Nothing

Description Displays a message box stating **`text`** to the user. Like message boxes in most operating systems/GUI display libraries, the message box is modal, and must be dismissed by clicking "OK" before any further interaction with Ərk is possible.

`directory`

Arguments None

Returns Str

Description Returns the directory the plugin is located in. If the plugin is in a single file, this will return the location of the plugin directory; if the plugin is in a package, this will return the package's directory.

Plugin Packages

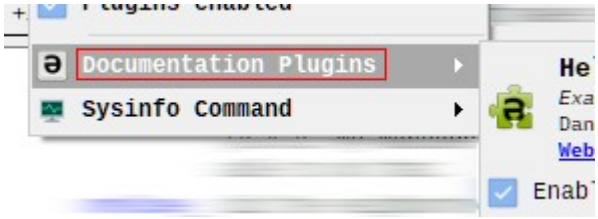
Plugins should be in their own directory in the "plugins" directory; this directory name can be any valid string for a Python module. This directory/plugin combination is called a "package".

For example, let's assume you have an Ərk plugin named "myplugin.py". You've decided that you want to use "dans_plugins" as your package name. Create a directory in your Ərk plugin directory named "dans_plugins", and copy/mode "myplugin.py" into the new directory. Your plugin package now looks like:

```
dans_plugins
  •  myplugin.py
```

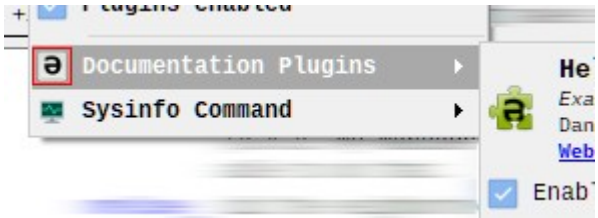
This is all you need for a plugin package. However, how the package (and the plugins it contains) are displayed in the Ərk GUI can be further customized.

Package Name



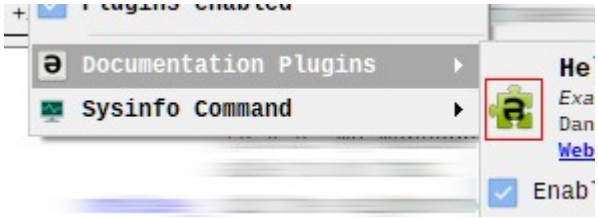
The plugin's Python name is displayed as the package's name by default. If you'd like to customize this (for example, change the package's name to include punctuation not normally allowed by Python), create a file named "package.txt" (or just "package") in the root directory of the package. Write the new name in this file as the first and only line. Using the above example, if you wanted your "dans_plugin" package to be displayed as "Dan's Erk Plugins", you would write "Dan's Erk Plugins" in the package's "package.txt".

Package Icon



To create a custom icon for a package, create a PNG file with the desired icon image and place it in the root directory of the package; change the package icon's filename to "package.png". This icon will be displayed in Ərk's "Plugins" menu. Optional; if omitted, the default package icon will be used.

Plugin Icons



An icon can be specified for a plugin; this is the icon that will be displayed in Ərk's "Plugins" menu. To set an icon, place a PNG with the same name as the plugin's class file name with the extension ".py" replaced with ".png" in the same directory as the plugin's package. The icon should be 25 pixels by 25 pixels to be displayed properly. Optional; if omitted, the default plugin icon will be used.

Use Ərk to generate a "blank" plugin

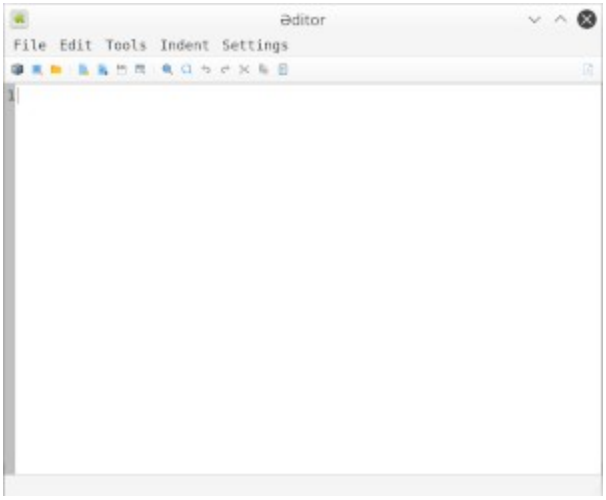
The Ərk client has a command-line flag that will generate a "blank" plugin package, ready to be edited. It will create a directory with a Python-safe name in the user's current directory, and place all the files needed for a plugin in it.

```
C:\> python erk.py --generate "My brand new plugin"
Creating plugin package Mybrandnewplugin…
Done!
C:\>
```

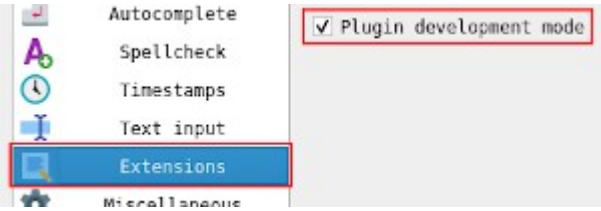
The new package will not load into Ərk; it must be edited. It includes a plugin source code skeleton ("plugin.py"), a package name file ("package.txt"), and default icons for both the package and plugin.

Once edited, simply copy the package directory into Ərk's plugin directory.

Əditor, the Ərk Plugin Editor



Ərk has a built-in editor and development environment for writing plugins. By default, it is hidden; to make the editor accessible, open the "Preferences" menu, and open the “Extensions” section. Turn on “Plugin development mode”.

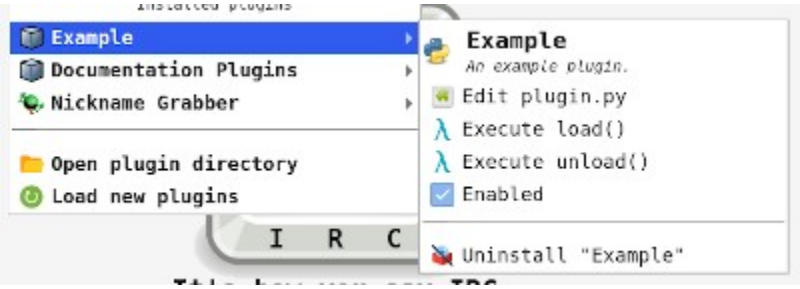


The next time the "Plugins" menu is opened, it will feature a number of new options. The features will remain available until "Development mode" is turned off.

To start the editor, click on "Əditor". The editor features colored syntax highlighting for Python code, and many of the features a code-oriented text editor would feature.

Development mode

Development mode also activates a number of features to make plugin development easier. Every plugin entry in the "Plugins" menu gets new options:



- **Edit**. Click to edit the plugin's source file in the editor.
- **Execute `load()`**. This will force the plugin's **`load()`** method to execute (if the plugin has a **`load()`** method).
- **Execute `unload()`**. This will force the plugin's **`unload()`** method to execute (if the plugin has an **`unload()`** method).
- **Enabled.** Whether the plugin is enabled or not. If this is not selected, the plugin will not load or execute.

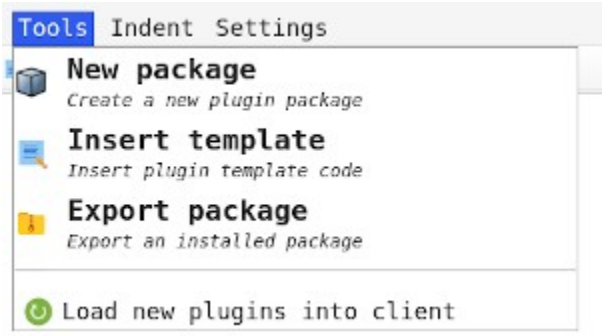
The main "Plugins" menu also gets five new options:

- **Editor**. This opens the code editor.
- **Install.** Plugins can be installed into Ərk from a zip file.
- **Export**. Any installed plugins can be exported to a zip file; this file an then be used with the “Install” option.
- **Open plugin directory**. This will open Ərk's plugin directory in the computer's file explorer.
- **Load new plugins**. This will load any new plugins, into memory. If the plugin hasn't been loaded, the plugin's **`load()`** method will be executed.

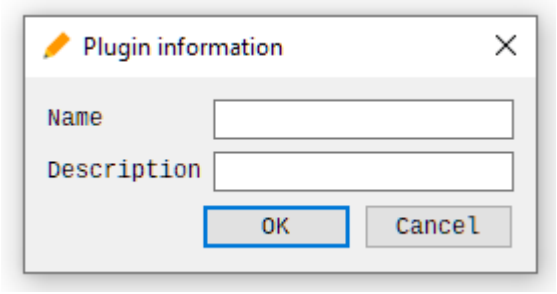
Features

The editor features almost everything you need to create plugins and plugin packages for Ərk easily. These tools include a package creator, a tool to export installed plugin packages, and a plugin code generator.

All three of these features can be found in the "Tools" menu:



- **New package**. This brings up a dialog asking the user for a package name and description:



This information is used to create an all-new plugin package in Ərk's plugin directory. This includes plugin and package icons, "package.txt", and a new plugin source file (named "plugin.py"); this file is opened in the editor, ready for editing.

- **Insert template**. This brings up a dialog asking for plugin information, and uses that data to generate a "blank" plugin, which is inserted wherever the cursor is in the editor.

- **Export package**. This will bring up a dialog allowing the user to select any installed package; the selected package will be placed in a zip archive and saved. This archive can now be installed with the "Install plugin" menu option in Ərk's "Plugins" menu.

Examples

"Hello, world!" plugin

This plugin will implement new command: "/hello". This will display the traditional "Hello, world!" message in the window where the command was entered.

```
 1 from erk import *
 2
 3 class HelloWorld(Plugin):
 4      def __init__(self):
 5           self.name = "Hello World plugin"
 6           self.description = "Example plugin for documentation"
 7
 8      def input(self,client,name,text):
 9
10           # Look for our new command
11           if text=="/hello":
12
13                # Found it! Now lets display our message
14                self.print("Hello, world!")
15
16                # Now, we return "True" to make sure that
17                # "/hello" isn't sent to the IRC server as
18                # a chat messages
19                return True
```

Note taking plugin

This plugin will introduce three new commands: "/note" (for adding a note), "/notes" (for displaying all stored notes), and "/clear" (for deleting all stored notes).

```
from erk import *

class Notes(Plugin):
     def __init__(self):
          self.name = "Note taking plugin"
          self.description = "Example plugin for documentation"
          self.notes = []

     def input(self,client,name,text):

          # Tokenize the input
          tokens = text.split()

          # Handle the "/clear" command
          # This will delete any stored notes
          if len(tokens)>0 and tokens[0].lower()=="/clear":
               self.notes = []
               return True

          # Handle the "/note" command
          # This adds a new note to the stored notes
          if len(tokens)>0 and tokens[0].lower()=="/note":
               tokens.pop(0)
               n = ' '.join(tokens)
               self.notes.append(n)
               return True

          # Handle the "/notes" command
          # This will display any stored notes
          if len(tokens)>0 and tokens[0].lower()=="/notes":

               # If there are no stored notes, let the
               # user know and return
               if len(self.notes)==0:
                    self.print("No notes found")
                    return True

               # Format the note list using HTML
               t = "<ul>"
               for n in self.notes:
                    t = t + "<li>"+n+"</li>"
               t = t + "</ul>"

               # Display the stored notes to the user
               self.print(t)
```