GeanyPy Documentation

Release 1.0

Matthew Brush <mbrush@codebrainz.ca>

February 17, 2017

Contents

Contents:

CHAPTER 1

Introduction

GeanyPy allows people to write their Geany plugins in Python making authoring a plugin much more accessible to non C programmers. What follows is a description of installing and using the GeanyPy plugin, paving the way for the rest of the documentation to covert the details of programming with the GeanyPy bindings of the Geany API.

CHAPTER 2

Installation

Currently there are no binary packages available for installing GeanyPy so it must be installed from source. The following instructions will describe how to do this.

Getting the Source

The best way currently to get GeanyPy is to check it out from it's repository on GitHub.com. You can clone GeanyPy's *master* branch by issuing the following command in a directory where you want to store its source code:

```
$ git clone git://github.com/codebrainz/geanypy.git
$ cd geanypy
```

Alternatively, you can download the *master* branch compressed into a tarball or zip file. Then extract it where you want to store GeanyPy's source, for example:

```
$ cd ~/src
$ wget -O geanypy.tar.gz https://github.com/codebrainz/geanypy/tarball/master
$ tar xf geanypy.tar.gz
$ cd codebrainz-geanypy-*
```

The first method using Git is the best, since it allows you to update your copy of GeanyPy quite easily and also makes it easier to contribute back to the GeanyPy project if you want to.

Dependencies and where to get them

Of course depending on what operating system and distribution you're using, getting setup for this process may vary wildly. At present, the following dependencies are required to compile GeanyPy:

GCC, Autotools, and all the usual build tools

For example on Debian (Ubuntu, Mint, etc.) do:

```
$ apt-get install build-essential
```

Or on Fedora, something like this should do:

```
$ yum groupinstall "Development Tools" "Legacy Software Development"
```

The latest development version of Geany (0.21+)

Since GeanyPy is wrapping the current development version of Geany, to use it you are required to use that version of Geany. Until the next Geany release, you must either checkout the source code from Geany's Subversion repository or Git mirror or you can get one of the Nightly builds if you'd rather not compile it yourself.

For more information on installing Geany, please refer to Geany's excellent manual

Grabbing the dependencies for Geany on a Debian-based disto could be similar to this:

```
$ apt-get install libgtk2.0-0 libgtk2.0-0-dev
```

Or you might even have better luck with:

```
$ apt-get build-dep geany
```

A quick session for installing Geany on a Debian-based distro might look something like this:

```
$ cd ~/src
$ git clone http://git.geany.org/git/geany
$ cd geany
$ ./autogen.sh
$ ./configure
$ make
$ make install # possibly as root
```

By default, Geany will install into */usr/local* so if you want to install it somewhere else, for example */opt/geany*, then you would run the *configure* command above with the *prefix* argument, like:

```
$ ./configure --prefix=/opt/geany
```

It's important when installing GeanyPy later that you configure it with the same prefix where Geany is installed, otherwise Geany won't find the GeanyPy plugin.

Python 2.X and development files

As GeanPy makes use of Python's C API to gain access to Geany's C plugin API, both Python and the development files are required to compile GeanyPy. In theory, any Python version >= 2.6 and < 3.0 should be compatible with GeanyPy. You can download Python from its website or you can install the required packages using your distro's package manager, for example with Debian-based distros, run this:

```
$ apt-get install python python-dev
```

Note: Python 3.0+ is not supported yet, although at some point in the future, there are plans support it.

PyGTK and development files

Since Geany uses GTK+ as it's UI toolkit, GeanyPy uses PyGTK to interact with Geany's UI. You can either download PyGTK from it's website or you can install it with your system's pacakge manager, for example in Debian distros:

```
$ apt-get install python-gtk2 python-gtk2-dev
```

Note: Although PyGTK is all but deprecated (or is completely deprecated?) in favour of the newer and shinier PyGobject/GObject-introspection, it is still used in new code in GeanyPy due to lack of documentation and pacakge support for the newer stuff.

One fell swoop

If you're running a Debian-based distro, you should be able to install all the required dependencies, not including Geany itself, with the following command (as root):

```
$ apt-get install build-essential libgtk2.0-0 libgtk2.0-dev \
    python python-dev python-gtk2 python-gtk2-dev
```

And finally ... installing GeanyPy

Once all the dependencies are satisfied, installing GeanyPy should be pretty straight-forward, continuing on from *Getting the Source* above:

```
$ ./autogen.sh
$ ./configure --prefix=/the/same/prefix/used/for/geany
$ make
$ make install # possibly as root
```

CHAPTER 3

Getting Started

Before diving into the details and API docs for programming plugins with GeanyPy, it's important to note how it works and some features it provides.

What the heck is GeanyPy, really?

GeanyPy is a proxy plugin. Geany initially sees GeanyPy as any other plugin, but GeanyPy registers some additional stuff that enables Geany to load python plugins through GeanyPy. So to activate, use Geany's Plugin Manager under the Tools menu as you would for any other plugin.

Once the GeanyPy plugin has been activated, Geany should rescan the plugin directories and pick those up that are supported through GeanyPy. It'll integrate the python plugins into the Plugin Manager in an additional hierarchy level below GeanyPy.

- [ ] Geany plugin 1
- [x] GeanyPy
- [ ] Python plugin 1
- [x] Python plugin 2
- [ ] Python plugin 3
- [ ] Geany plugin 3

Remember that Geany looks in three places for plugins:

1. For system-wide plugins, it will search in (usually) /usr/share/geany or /usr/local/share/geany.
2. In Geany's config directory under your home directory, typically ~/.config/geany/plugins.
3. A user-configurable plugin directory (useful during plugin development).

Python Console

Another pretty cool feature of GeanyPy is the Python Console, which similar to the regular Python interactive interpreter console, but it's found in the Message Window area (bottom) in Geany. The *geany* Python module used to interact with Geany will be pre-imported for you, so you can mess around with Geany using the console, without ever having to even write a plugin.

Credits: The Python Console was taken, almost in its entirety, from the medit text editor. Props to the author(s) for such a nice piece of source code

Future Plans

Some time in the near future, there should be support for sending text from the active document into the Python Console. It will also be possible to have the Python Console either in a separate window, in the sidebar notebook or in the message window notebook.

Also, either integration with Geany's keybindings UI under the preferences dialog or a separate but similar UI just for Python plugins will be added. Currently using keybindings requires a certain amount of hackery, due to Geany expecting all plugins to be shared libraries written in C.

CHAPTER 4

Writing a Plugin - Quick Start Guide

This is just a quick tutorial to describe writing a GeanyPy compatible plugin in Python. Writing a plugin should be pretty straightforward for any Python programmers, especially those familiar with writing regular C plugins for Geany.

To illustrate the similarities to the C API, the example at the end of this section will create the same plugin as in Geany's Plugin Howto, except obviously written in Python.

The Plugin Interface

The first thing any plugin will want to do is to import the *geany* module:

```
import geany
```

Note: Due to the way the Geany plugin framework works, importing the *geany* module will certain fail if you just try running it standalone, outside of Geany/GeanyPy.

After that, you create a regular Python class which inherits from the *geany.Plugin* class:

```
import geany

class HelloWorld(geany.Plugin):
    pass
```

This will allow GeanyPy's Python Plugin Manager to locate the plugin as a GeanyPy plugin. If it doesn't inherit from *geany.Plugin* it will not be detected.

There are a few more parts of the interface that must be implemented in order for the plugin to be detected by GeanyPy:

```
import geany

class HelloWorld(geany.Plugin):

    __plugin_name__ = "HelloWorld" # required
    __plugin_version__ = "version of your plugin"
    __plugin_description__ = "description of your plugin"
    __plugin_author__ = "Your Name <your email address>"
```

These allow the Python Plugin Manager to glean information about your plugin which will be shown in the managers plugin list. All but the *__plugin_name__* attributes are optional, though recommended.

The next thing that's needed is an entry-point to the plugin. Since Python classes have a initialization method already, this seems like a logical entry-point:

```
import geany

class HelloWorld(geany.Plugin):

    __plugin_name__ = "HelloWorld" # required
    __plugin_version__ = "version of your plugin"
    __plugin_description__ = "description of your plugin"
    __plugin_author__ = "Your Name <your email address>"

    def __init__(self):
        do_stuff_when_loaded()
```

If you have some de-initialization code that needs to be run, you can add a *cleanup* method to the class that is guaranteed to be called when your plugin is unloaded, however it's optional:

```
import geany

class HelloWorld(geany.Plugin):

    __plugin_name__ = "HelloWorld" # required
    __plugin_version__ = "version of your plugin"
    __plugin_description__ = "description of your plugin"
    __plugin_author__ = "Your Name <your email address>"

    def __init__(self):
        do_stuff_when_loaded()

    def cleanup(self):
        do_stuff_when_unloaded()
```

And there you have it! That's the minimum requirements for writing a plugin that will be detected by GeanyPy. Ok, it doesn't do anything yet, but it will be shown in the Python Plugin Manager and can be loaded and unloaded.

Real-world Example

To put it into context, here's a plugin that mimics the plugin in Geany's Plugin Howto:

```
import gtk
import geany

class HelloWorld(geany.Plugin):

    __plugin_name__ = "HelloWorld"
    __plugin_version__ = "1.0"
    __plugin_description__ = "Just another tool to say hello world"
    __plugin_author__ = "John Doe <john.doe@example.org>"

    def __init__(self):
        self.menu_item = gtk.MenuItem("Hello World")
        self.menu_item.show()
        geany.main_widgets.tools_menu.append(self.menu_item)
        self.menu_item.connect("activate", self.on_hello_item_clicked)

    def cleanup(self):
        self.menu_item.destroy()
```

```
    def on_hello_item_clicked(widget, data):
        geany.dialogs.show_msgbox("Hello World")
```

Hopefully this makes it clear how to write a Python plugin using GeanyPy. This sample plugin is provided with GeanyPy if you want to try it out.

Logging

GeanyPy provides a logging adapter to log from Python to GLib's logging system which enables plugins to log messages to Geany's Help->Debug Messages window. To use it, simply call the base class' constructor in your __init__() method and use the new *self.logger* attribute. The logger attribute of *geany.Plugin* emulates a Python *logging.Logger* object and provides its logging functions:

- **log** (*lvl, msg, *args, **kwargs*)
- **debug** (*msg, *args, **kwargs*)
- **info** (*msg, *args, **kwargs*)
- **message** (*msg, *args, **kwargs*)
- **warning** (*msg, *args, **kwargs*)
- **error** (*msg, *args, **kwargs*)
- **exception** (*msg, *args, **kwargs*)
- **critical** (*msg, *args, **kwargs*)

The function *message* maps to the GLib log level *G_LOG_LEVEL_MESSAGE'which does not exist in Python but still can be used. The keyword argument 'exc_info* is supported for all logger methods with the same semantics as in Python (see Python logging documentation for details). However, the keyword argument *extra* is not supported and will be ignored. Since you cannot define your own formatter for the GLib logging system, passing the *extra* keyword argument does not make any sense.

Here is an example:

```
import geany

class HelloWorldLogger(geany.Plugin):

    __plugin_name__ = "HelloWorldLogger"
    __plugin_version__ = "1.0"
    __plugin_description__ = "Just another tool to log hello world"
    __plugin_author__ = "John Doe <john.doe@example.org>"

    def __init__(self):
        geany.Plugin.__init__(self)
        self.logger.info(u'Hello World')

    def cleanup(self):
        self.logger.debug(u'Bye Bye from HelloWorldLogger')
```

CHAPTER 5

API Documentation

GeanyPy's API mimics quite closely Geany's C plugin API. The following documentation is broken down by file/module:

The *geany* modules:

The app module

This modules contains a class to access application settings.

App Objects

class app.**App**

This class is initialized automatically and by the *geany* module and shouldn't be initalized by users. An instance of it is available through the *geany.app* attribute of the *geany* module.

All members of the *App* are read-only properties.

App.**configdir**
User configuration directory, usually ~/.config/geany. To store configuration files for your plugin, it's a good idea to use something like this:

```
conf_path = os.path.join(geany.app.configdir, "plugins", "yourplugin",
                         "yourconfigfile.conf")
```

App.**debug_mode**
If True, debug messages should be printed. For example, if you want to make a `print()` function that only prints when *App.debug_mode* is active, you could do something like this:

```
def debug_print(message):
    if geany.app.debug_mode:
        print(message)
```

App.**project**
If not `None`, the a `project.Project` for currently active project.

The `dialogs` module

This module contains some help functions to show file-related dialogs, miscellaneous dialogs, etc. You can of course just use the `gtk` module to create your own dialogs as well.

`dialogs.`**`show_input`**(*[title=None[, parent=None[, label_text=None[, default_text=None]]]]*)
Shows a `gtk.Dialog` to ask the user for text input.

Parameters

- **`title`** – The window title for the dialog.
- **`parent`** – The parent `gtk.Window` for the dialog, for example `geany.main_widgets.window`.
- **`label_text`** – Text to put in the label just about the input entry box.
- **`default_text`** – Default text to put in the input entry box.

Returns A string containing the text the user entered.

`dialogs.`**`show_input_numeric`**(*[title=None[, label_text=None[, value=0.0[, minimum=0.0[, maximum=100.0[, step=1.0]]]]]]*)
Shows a `gtk.Dialog` to ask the user to enter a numeric value using a `gtk.SpinButton`.

Parameters

- **`title`** – The window title for the dialog.
- **`label_text`** – Text to put in the label just about the numeric entry box.
- **`value`** – The initial value in the numeric entry box.
- **`minimum`** – The minimum allowed value that can be entered.
- **`maximum`** – The maximum allowed value that can be entered.
- **`step`** – Amount to increment the numeric entry when it's value is moved up or down (ex, using arrows).

Returns The value entered if the dialog was closed with ok, or `None` if it was cancelled.

`dialogs.`**`show_msgbox`**(*text[, msgtype=gtk.MESSAGE_INFO]*)
Shows a `gtk.Dialog` to show the user a message.

Parameters

- **`text`** – The text to show in the message box.
- **`msgtype`** – The message type which is one of the Gtk Message Type Constants.

`dialogs.`**`show_question`**(*text*)
Shows a `gtk.Dialog` to ask the user a Yes/No question.

Parameters **`text`** – The text to show in the question dialog.

Returns `True` if the Yes button was pressed, `False` if the No button was pressed.

`dialogs.`**`show_save_as`**()
Shows Geany's *Save As* dialog.

Returns `True` if the file was saved, `False` otherwise.

The `document` module

This module provides functions for working with documents. Most of the module-level functions are used for creating instances of the *Document* object.

`document.`**`find_by_filename`**(*filename*)
Finds a document with the given filename from the open documents.

Parameters **`filename`** – Filename of *Document* to find.

Returns A *Document* instance for the found document or `None`.

`document.`**`get_current`**()
Gets the currently active document.

Returns A *Document* instance for the currently active document or `None` if no documents are open.

`document.`**`get_from_page`**(*page_num*)
Gets a document based on it's `gtk.Notebook` page number.

Parameters **`page_num`** – The tab number of the document in the documents notebook.

Returns A *Document* instance for the corresponding document or `None` if no document matched.

`document.`**`index`**(*index*)
Gets a document based on its index in Geany's documents array.

Parameters **`index`** – The index of the document in Geany's documents array.

Returns A *Document* instance for the corresponding document or `None` if not document matched, or the document that matched isn't valid.

`document.`**`new_file`**([*filename=None*[, *filetype=None*[, *text=None*]]])
Creates a document file.

Parameters

- **`filename`** – The documents filename, or `None` for *untitled*.
- **`filetype`** – The documents filetype or `None` to auto-detect it from *filename* (if it's not `None`)
- **`text`** – Initial text to put in the new document or `None` to leave it blank

Returns A *Document* instance for the new document.

`document.`**`open_file`**(*filename*[, *read_only=False*[, *filetype=None*[, *forced_enc=None*]]])
Open an existing document file.

Parameters

- **`filename`** – Filename of the document to open.
- **`read_only`** – Whether to open the document in read-only mode.
- **`filetype`** – Filetype to open the document as or `None` to detect it automatically.
- **`forced_enc`** – The file encoding to use or `None` to auto-detect it.

Returns A *Document* instance for the opened document or `None` if it couldn't be opened.

`document.`**`open_files`**(*filenames*, *read_only=False*, *filetype=""*, *forced_enc=""*)
Open multiple files. This actually calls *open_file()* once for each filename in *filenames*.

Parameters

- **filenames** – List of filenames to open.
- **read_only** – Whether to open the document in read-only mode.
- **filetype** – Filetype to open the document as or `None` to detect it automatically.
- **forced_enc** – The file encoding to use or `None` to auto-detect it.

`document.`**remove_page**(*page_num*)

Remove a document from the documents array based on it's page number in the documents notebook.

Parameters **page_num** – The tab number of the document in the documents notebook.

Returns `True` if the document was actually removed or `False` otherwise.

`document.`**get_documents_list**()

Get a list of open documents.

Returns A list of *`Document`* instances, one for each open document.

`Document` Objects

class `document.`**Document**

The main class holding information about a specific document. Unless otherwise noted, the attributes are read-only properties.

basename_for_display

The last part of the filename for this document, possibly truncated to a maximum length in case the filename is very long.

notebook_page

The page number in the `gtk.Notebook` containing documents.

status_color

Gets the status color of the document, or `None` if the default widget coloring should be used. The color is red if the document has changes, green if it's read-only or `None` if the document is unmodified but writable. The value is a tuple of the RGB values for red, green, and blue respectively.

encoding

The encoding of this document. Must be a valid string representation of an encoding. This property is read-write.

file_type

The file type of this document as a `Filetype` instance. This property is read-write.

text_changed

Whether this document's text has been changed since it was last saved.

file_name

The file name of this document.

has_bom

Indicates whether the document's file has a byte-order-mark.

has_tags

Indicates whether this document supports source code symbols (tags) to show in the sidebar.

index

Index of the document in Geany's documents array.

is_valid

Indicates whether this document is active and all properties are set correctly.

read_only
Whether the document is in read-only mode.

real_path
The link-dereferenced, locale-encoded file name for this document.

editor
The `Editor` instance associated with this document.

close()
Close this document.

Returns `True` if the document was closed, `False` otherwise.

reload([*forced_enc=None*])
Reloads this document.

Parameters **forced_enc** – The encoding to use when reloading this document or `None` to auto-detect it.

Returns `True` if the document was actually reloaded or `False` otherwise.

rename(*new_filename*)
Rename this document to a new file name. Only the file on disk is actually renamed, you still have to call *save_as()* to change the document object. It also stops monitoring for file changes to prevent receiving too many file change events while renaming. File monitoring is setup again in *save_as()*.

Parameters **new_filename** – The new filename to rename to.

save([*force=False*])
Saves this documents file on disk.

Saving may include replacing tabs by spaces, stripping trailing spaces and adding a final new line at the end of the file, depending on user preferences. Then, the *document-before-save* signal is emitted, allowing plugins to modify the document before it's saved, and the data is actually written to disk. The file type is set again or auto-detected if it wasn't set yet. Afterwards, the *document-save* signal is emitted for plugins. If the file is not modified, this method does nothing unless *force* is set to `True`.

Note: You should ensure that *file_name* is not `None` before calling this; otherwise call *dialogs.show_save_as()*.

Parameters **force** – Whether to save the document even if it's not modified.

Returns `True` if the file was saved or `False` if the file could not or should not be saved.

save_as(*new_filename*)
Saves the document with a new filename, detecting the filetype.

Parameters **new_filename** – The new filename.

Returns `True` if the file was saved or `False` if it could not be saved.

The geany package and module

All of GeanyPy's bindings are inside the *geany* package which also contains some stuff in it's `__init__` file, acting like a module itself.

geany.**app**
An instance of *app.App* to store application information.

geany.**main_widgets**
An instance of `mainwidgets.MainWidgets` to provide access to Geany's important GTK+ widgets.

geany.**signals**
An instance of signalmanager.SignalManager which manages the connection, disconnection, and emission of Geany's signals. You can use this as follows:

```
geany.signals.connect('document-open', some_callback_function)
```

geany.**is_realized**()
This function, which is actually in the geany.main module will tell you if Geany's main window is realized (shown).

geany.**locale_init**()
Again, from the geany.main module, this will initialize the *gettext* translation system.

geany.**reload_configuration**()
Also from the geany.main module, this function will cause Geany to reload most if it's configuration files without restarting.

Currently the following files are reloaded:

- all template files
- new file templates
- the New (with template) menus will be updated
- Snippets (snippets.conf)
- filetype extensions (filetype_extensions.conf)
- *settings* and *build_settings* sections of the filetype definition files.

Plugins may call this function if they changed any of these files (e.g. a configuration file editor plugin).

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

a

d

g

T