Tweet

Jun 17, 2012

Asynchronously Read Files in Python with Gio

Most people will learn how to read and write to files in Python using the built-in file objects. That works great for simple read/write operations on the local filesystem. However, if you need a more advanced I/O library, take a look at Gio.

Gio provides an abstract I/O API without having to know what the underlying filesystem is. In other words, you can read files from various sources including http, ftp, ssh, etc. The Gio library has support for asynchronous operations and access to a ton of other useful information (mime types, themed icon names, etc.).

Here is a closer look at asynchronously reading files in Python using Gio.

A Quick Note about GObject Introspection

These examples are based on PyGObject in which the python code is accessing the underlying C libraries through a little bit of gobject introspection magic. Therefore the API documentation is for C but it's pretty easy to understand how to translate the C documentation to Python.

The GioFile Object

The `Gio.File` object is the primary file and directory interface you work with when using the Gio library. You can use `Gio.File` objects to get all sorts of interesting information about a file and perform various operations on the file object. They can be created from a URI, a local filesystem path, and command line arguments. I'll be using the URI to create `Gio.File` objects.

Creating Gio.File Objects

```
from gi.repository

giofile = Gio.File.new_for_uri('http://www.micahcarrick.com')
giofile = Gio.File.new_for_path('/home/micah/')
giofile = Gio.File.new_for_commandline_arg(sys.argv[1])
```

Simple Asynchronous Read

The easiest way to asynchronously read a file is to use the `load_contents_async()` method. Since the file is read asynchronously, a callback function is defined to be called when the file is finished loading. The arguments passed to this callback function are defined by GAsyncReadyCallback.

In the callback function, a call to `load_contents_finish()` finishes the asynchronous read and returns a 3-tuple containing a boolean if the operation was a success, the contents from the read, and the file's etag.

In the example below, a GLib main event loop is used since this program doesn't have a GUI. It is necessary to have a main event loop otherwise the program would likely get to the end and terminate before the asynchronous read has completed.

```
#!/usr/bin/env python
from gi.repository import GLib, Gio

class GioAsyncReadExample(object):

    def read(self, uri):
        print "Reading %s" % uri
        giofile = Gio.File.new_for_uri(uri)
        giofile.load_contents_async(None, self._load_contents_cb, None)

    def _load_contents_cb(self, giofile, result, user_data=None):
        success, contents, etag = giofile.load_contents_finish(result)
        print "Finished reading %d bytes" % len(contents)
        loop.quit()

if __name__ == "__main__":
    loop = GLib.MainLoop()
```

Contact Me

Categories

Android Programming
Django
Game Programming
GNOME
GTK+ Programming
Linux
Python Programming
Robotics & Electronics
Web Development

Recent Posts

SSL behind an Nginx Reverse Proxy in Django 1.4
Using libgdx without Eclipse
Django Project Plugin 3.1.3 for Gedit
A BASE_URL Template Variable in Django
Setting USB Permissions for USBtinyISP in Fedora
CadSoft Eagle 6.2 in Fedora 17 x86_64
HP Pavillion dv2000 Motherboard Fix
Canon MP560 in Fedora
Close Buttons on Gtk.Notebook Tabs
Simple "Hello World" in Python with GTK+ 3

```
    reader = GioAsyncReadExample()
    reader.read("http://www.w3.org/TR/1999/REC-html401-19991224/html40.txt")
    print "Entering GLib main loop."
    loop.run()
```

This program will asynchronously read the contents of the HTML 4.01 Specification from the W3C website, display how many bytes it read, and then break out of the main event loop to terminate the appliaction. The output looks like this:

```
[micah@octopus Desktop]$ ./load_contents_async.py
Reading http://www.w3.org/TR/1999/REC-html401-19991224/html40.txt
Entering GLib main loop
Finished reading 792284 bytes
```

Error Handling

There are plenty of things that can go wrong when reading a file. Does it exist? What if it is deleted in the middle of reading it? What if the internet goes down during a remote file read? What if you don't have permission to read it? The list goes on and on.

You can catch a `GLib.GError` exception when calling `load_contents_finish()` to handle the errors.

```
try:
    success, contents, etag = giofile.load_contents_finish(result)
    print "Finished reading %d bytes" % len(contents)
except GLib.GError as error:
    print str(error)
```

You can handle specific errors by checking for a specific 'Gio.IOErrorEnum' constant.

```
try:
    success, contents, etag = giofile.load_contents_finish(result)
    print "Finished reading %d bytes" % len(contents)
except GLib.GError as error:
    if error.code == Gio.IOErrorEnum.PERMISSION_DENIED:
        print "Sorry dude, you're not allowed to read this file."
    elif error.code == Gio.IOErrorEnum.IS_DIRECTORY:
        print "C'mon man, you can't read a directory."
    else:
        print str(error)
```

Cancelling Asynchronous Operations

The first argument passed to `load_contents_async()` is a `Gio.Cancellable` object which provides the mechanism to cancel an asynchronous operation. In the simple example, `None` was passed and therefore there was no way to cancel the operation. However, in a GUI application in which these operations may take a long time (eg. a huge file over the network). Providing the end-user with a means to cancel the operation is one of the benefits of using asynchronous I/O.

A `Gio.Cancellable` object can be re-used for all of an application's asynchronous operations. It's `reset()` method is called before it is passed to an async function to make sure it's not already flagged as being cancelled. A button or key press within the user interface could call the `cancel` method of the `Gio.Cancellable` object at any time during the asynchronous operation.

The call to `load_contents_finish()` in the callback function would handle the cancellation of an async operation by catching the `GLib.Error` exception with the `Gio.IOErrorEnum.CANCELLED` code.

```
#!/usr/bin/env python
from gi.repository import GLib, Gio

class GioAsyncReadExample(object):
    def __init__(self):
        self._cancellable = Gio.Cancellable()

    def cancel(self):
        # This could be connected to the "clicked" signal of a Gtk.Button
        print "Canceling asynchronous operation..."
        self._cancellable.cancel()

    def read(self, uri):
        print "Reading %s" % uri
        giofile = Gio.File.new_for_uri(uri)
        self._cancellable.reset()
        giofile.load_contents_async(self._cancellable, self._load_contents_cb,

    def _load_contents_cb(self, giofile, result, user_data=None):
        try:
            success, contents, etag = giofile.load_contents_finish(result)
            print "Finished reading %d bytes" % len(contents)
```

```
        except GLib.GError as error:
            if error.code == Gio.IOErrorEnum.CANCELLED:
                print "Alright then, we've aborted the read operation."
            else:
                print str(error)

        loop.quit()

if __name__ == "__main__":
    loop = GLib.MainLoop()
    reader = GioAsyncReadExample()
    reader.read("http://www.w3.org/TR/1999/REC-html401-19991224/html40.txt")
    reader.cancel() # cancel the read in progress
    print "Entering GLib main loop"
    loop.run()
```

Character Encoding

The contents returned from these read operations are not converted to any particular character encoding for you. When working with GTK+ most of the calls will be expecting UTF-8 text. Python's `decode`, `encode`, and `unicode` can be used to convert to and from various character sets.

```
try:
    decoded = contents.decode("UTF-8")
except UnicodeDecodeError:
    print "Error: Unknown character encoding. Expecting UTF-8"
```

Auto-detecting character encoding is way beyond the scope of this post (and is technically impossible), but, you could take a look at the Python Unicode HOWTO to get started. There are some libraries that do some advanced character encoding detection. A "poor-man's" approach is to simply try to decode using some common encodings.

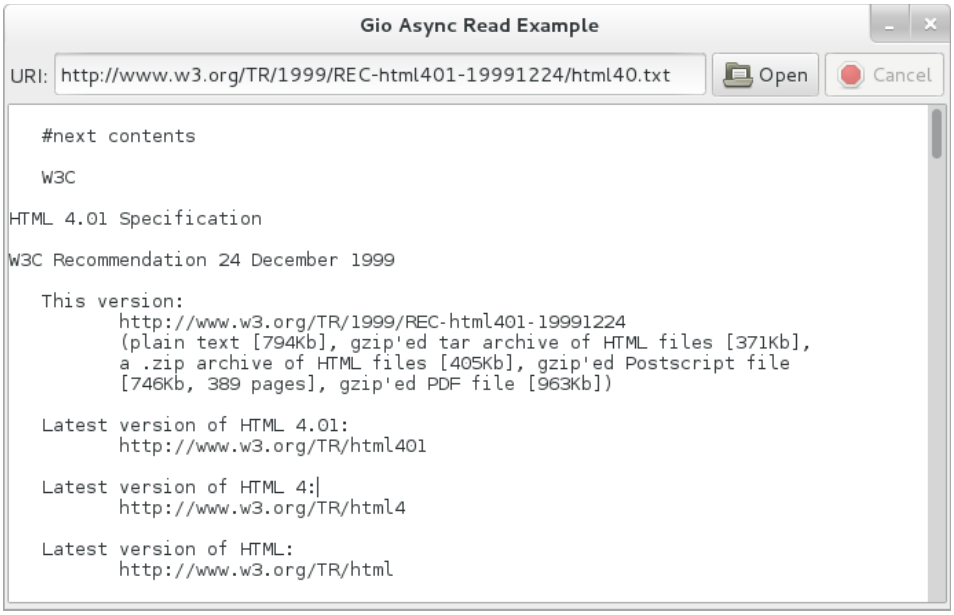
```
encodings = ['UTF-8', 'ISO-8859-15']
document_encoding = None

for encoding in encodings:
    try:
        decoded = contents.decode(encoding)
        document_encoding = encoding
        print "Auto-detected encoding as %s" % encoding
        break
    except UnicodeDecodeError:
        pass
if not document_encoding:
    print "Unknown character encoding"
```

Putting It All Together

These concepts can be put together to build a simple GTK+ application which can asynchronously read files, including remote files, into a `Gtk.TextView`. A button allows the user to cancel the operation if it is taking too long. The character encoding of the document is detected (guessed) and encoded to UTF-8 as requried by the `Gtk.TextView`.

A `Gtk.MessageDialog` displays any errors that come back from the `Gio` library.

If you run this application in a terminal and open `http://www.w3.org/TR/1999/REC-html401-19991224/html40.txt` you will see that the "detected" character encoding is is `ISO-8859-15`. If you then load my website `http://www.micahcarrick.com` the character encoding will be detected as `UTF-8`.




```python
#!/usr/bin/env python
from gi.repository import GLib, Gio, Pango, Gtk

class GioGtkExample(object):
    def __init__(self):
        self._cancellable = Gio.Cancellable()
        self._create_window()

    def run(self):
        """ Show window and enter GTK+ main event loop """
        self.window.show()
        Gtk.main()

    def _reset(self):
        """ Reset the UI to the ready state """
        self._cancellable.reset()
        self._entry.set_sensitive(True)
        self._open_button.set_sensitive(True)
        self._cancel_button.set_sensitive(False)

    def read(self, uri):
        """ Read the specified URI into the Gtk.TextView """
        # toggle widgets sensitivity
        self._entry.set_sensitive(False)
        self._open_button.set_sensitive(False)
        self._cancel_button.set_sensitive(True)

        # clear text view
        self._view.get_buffer().set_text("")

        # begin read operation
        giofile = Gio.File.new_for_uri(uri)
        giofile.load_contents_async(self._cancellable, self._load_contents_cb,

    def _load_contents_cb(self, giofile, result, user_data=None):
        """ Callback for Gio.File.load_contents_async() """
```

What About Asynchronously Writing Files?

The concepts for asynchronously writing files is pretty much the same. In fact, the concept for pretty much all of the asynchronous file operations are the same. See the `replace_async()` and `append_to_async()` methods to asynchronously write a file.

Did you enjoy **Asynchronously Read Files in Python with Gio?** If you would like to help support my work, A donation of a buck or two would be very much appreciated.

Donate