

Application resources

Previous sections in this guide have described how to load images, media and text documents using `pyglet`. Applications also usually have the need to load other data files: for example, level descriptions in a game, internationalised strings, and so on.

Programmers are often tempted to load, for example, an image required by their application with:

```
image = pyglet.image.load('logo.png')
```

This code assumes `logo.png` is in the current working directory. Unfortunately the working directory is not necessarily the same as the directory containing the application script files.

- Applications started from the command line can start from an arbitrary working directory.
- Applications bundled into an egg, Mac OS X package or Windows executable may have their resources inside a ZIP file.
- The application might need to change the working directory in order to work with the user's files.

A common workaround for this is to construct a path relative to the script file instead of the working directory:

```
import os

script_dir = os.path.dirname(__file__)
path = os.path.join(script_dir, 'logo.png')
image = pyglet.image.load(path)
```

This, besides being tedious to write, still does not work for resources within ZIP files, and can be troublesome in projects that span multiple packages.

The `pyglet.resource` module solves this problem elegantly:

```
image = pygame.resource.image('logo.png')
```

The following sections describe exactly how the resources are located, and how the behaviour can be customised.

Loading resources

Use the `pygame.resource` module when files shipped with the application need to be loaded. For example, instead of writing:

```
data_file = open('file.txt')
```

use:

```
data_file = pygame.resource.file('file.txt')
```



There are also convenience functions for loading media files for pygame. The following table shows the equivalent resource functions for the standard file functions.

File function	Resource function	Type
<code>open</code>	<code>pyglet.resource.file()</code>	File-like object
<code>pyglet.image.load()</code>	<code>pyglet.resource.image()</code>	<code>Texture</code> OR <code>TextureRegion</code>
<code>pyglet.image.load()</code>	<code>pyglet.resource.texture()</code>	<code>Texture</code>
<code>pyglet.image.load_animation()</code>	<code>pyglet.resource.animation()</code>	<code>Animation</code>
<code>pyglet.media.load()</code>	<code>pyglet.resource.media()</code>	<code>Source</code>
<code>pyglet.text.load()</code> mimetype = <code>text/plain</code>	<code>pyglet.resource.text()</code>	<code>UnformattedDocument</code>
<code>pyglet.text.load()</code> mimetype = <code>text/html</code>	<code>pyglet.resource.html()</code>	<code>FormattedDocument</code>
<code>pyglet.text.load()</code> mimetype = <code>text/vnd.pyglet-attributed</code>	<code>pyglet.resource.attributed()</code>	<code>FormattedDocument</code>
<code>pyglet.font.add_file()</code>	<code>pyglet.resource.add_font()</code>	<code>None</code>

`pyglet.resource.texture()` is for loading stand-alone textures. This can be useful when using the texture for a 3D model, or generally working with OpenGL directly.

`pyglet.resource.image()` is optimised for loading sprite-like images that can have their texture coordinates adjusted. The resource module attempts to pack small images into larger texture atlases (explained in [Texture bins and atlases](#)) for efficient rendering (which is why the return type of this function can be `TextureRegion`). It is also advisable to use the texture atlas classes directly if you wish to have different anchor points on multiple copies of the same image. This is because when loading an image more than once, you will actually get the **same** object back. You can still use the resource module for getting the image location, and described in the next section.

Resource locations

Some resource files reference other files by name. For example, an HTML document  **latest**  `` elements. In this case your application needs to locate `image.png` relative to the original HTML file.

Use `pyglet.resource.location()` to get a `Location` object describing the location of an application resource. This location might be a file system directory or a directory within a ZIP file. The `Location` object can directly open files by name, so your application does not need to distinguish between these cases.

In the following example, a `thumbnails.txt` file is assumed to contain a list of image filenames (one per line), which are then loaded assuming the image files are located in the same directory as the `thumbnails.txt` file:

```
thumbnails_file = pyglet.resource.file('thumbnails.txt', 'rt')
thumbnails_location = pyglet.resource.location('thumbnails.txt')

for line in thumbnails_file:
    filename = line.strip()
    image_file = thumbnails_location.open(filename)
    image = pyglet.image.load(filename, file=image_file)
    # Do something with `image`...
```

This code correctly ignores other images with the same filename that might appear elsewhere on the resource path.

Specifying the resource path

By default, only the script home directory is searched (the directory containing the `__main__` module). You can set `pyglet.resource.path` to a list of locations to search in order. This list is indexed, so after modifying it you will need to call `pyglet.resource.reindex()`.

Each item in the path list is either a path relative to the script home, or the name of a Python module preceded with an “at” symbol (`@`). For example, if you would like to package all your resources in a `res` directory:

```
pyglet.resource.path = ['res']
pyglet.resource.reindex()
```

Items on the path are not searched recursively, so if your resource directory itself has subdirectories, these need to be specified explicitly:

```
pyglet.resource.path = ['res', 'res/images', 'res/sounds', 'res/fonts']
pyglet.resource.reindex()
```

The entries in the resource path always use forward slash characters as path separators even when the operating systems using a different character.

Specifying module names makes it easy to group code with its resources. The following example uses the directory containing the hypothetical `gui.skins.default` for resources:

```
pyglet.resource.path = ['@gui.skins.default', '.']
pyglet.resource.reindex()
```

Multiple loaders

A `Loader` encapsulates a complete resource path and cache. This lets your application cleanly separate resource loading of different modules. Loaders are constructed for a given search path, and exposes the same methods as the global `pyglet.resource` module functions.

For example, if a module needs to load its own graphics but does not want to interfere with the rest of the application's resource loading, it would create its own `Loader` with a local search path:

```
loader = pyglet.resource.Loader(['@' + __name__])
image = loader.image('logo.png')
```

This is particularly suitable for “plugin” modules.

You can also use a `Loader` instance to load a set of resources relative to some user-specified document directory. The following example creates a loader for a directory specified on the command line:

```
import sys
home = sys.argv[1]
loader = pyglet.resource.Loader(script_home=[home])
```

This is the only way that absolute directories and resources not bundled with an application should be used with `pyglet.resource`.

Saving user preferences and data

Because Python applications can be distributed in several ways, including within ZIP files, it is usually not feasible to save user preferences, high score lists, and so on within the application directory (or worse, the working directory). The resource module provides functions for assisting with this.

The `pyglet.resource.get_settings_path()` function returns a directory suitable for writing configuration related data. The directory used follows the operating system's convention:

- `~/.config/ApplicationName/` on Linux (depends on `XDG_CONFIG_HOME` environment variable).
- `$HOME\Application Settings\ApplicationName` on Windows
- `~/Library/Application Support/ApplicationName` on Mac OS X

The `pyglet.resource.get_data_path()` function returns a directory suitable for writing arbitrary data, such as save files. The directory used follows the operating system's convention:

- `~/.local/share/ApplicationName/` on Linux (depends on `XDG_DATA_HOME` environment variable).
- `$HOME\Application Settings\ApplicationName` on Windows
- `~/Library/Application Support/ApplicationName` on Mac OS X

The returned directory names are not guaranteed to exist – it is the application's responsibility to create them. The following example opens a high score list file for a game called “SuperGame” into the data directory:

```
import os

dir = pyglet.resource.get_data_path('SuperGame')
if not os.path.exists(dir):
    os.makedirs(dir)
filename = os.path.join(dir, 'highscores.txt')
file = open(filename, 'wt')
```

Simplify infrastructure with MongoDB Atlas, the leading developer data platform

Ads by EthicalAds