Reloading Python Modules

October 24, 2010

Being able to reload code modules is one of the many nice features of Python. This allows developers to modify parts of a Python application while the interpreter is running. In general, all that needs to be done is pass a module object to the `imp.reload()` function (or just `reload()` in Python 2.x), and the module will be reloaded from its source file.

There are a few potential complications, however.

If any other code references symbols exported by the reloaded module, they may still be bound to the original code. For example, imagine if module `A` contains the constant `INTERVAL = 5`, and module `B` imports that constant into its namespace (`from A import INTERVAL`). If we change the constant to `INTERVAL = 10` and just reload module `A`, any values in module `B` that were based on `INTERVAL` won't be updated to reflect its new value.

The solution to this problem is to also reload module `B`. But it's important to only reload module `B` *after* module `A` has been reloaded. Otherwise, it won't pick up the updated symbols.

PyUnit deals with a variation of this problem by introducing a rollback importer. That approach "rolls back" the set of imported modules to some previous state by overriding Python's global `__import__` hook. PyUnit's solution is effective at restoring the interpreter's state to pre-test conditions, but it's not a general solution for live code reloading because the unloaded modules aren't automatically reloaded.

The following describes a general module reloading solution which aims to make the process automatic, transparent, and reliable.

Recording Module Dependencies

It is important to understand the dependencies between loaded modules so that they can be reloaded in the correct order. The ideal solution is to build a dependency graph as the modules are loaded. This can be accomplished by installing a custom import hook that is called as part of the regular module import machinery.

```
import builtins

_baseimport = builtins.__import__
_dependencies = dict()
_parent = None

def _import(name, globals=None, locals=None, fromlist=None, level=-1):
    # Track our current parent module.  This is used to find our current
    # place in the dependency graph.
    global _parent
    parent = _parent
    _parent = name
```

```
    # Perform the actual import using the base import function.
    m = _baseimport(name, globals, locals, fromlist, level)

    # If we have a parent (i.e. this is a nested import) and this is a
    # reloadable (source-based) module, we append ourself to our parent's
    # dependency list.
    if parent is not None and hasattr(m, '__file__'):
        l = _dependencies.setdefault(parent, [])
        l.append(m)

    # Lastly, we always restore our global _parent pointer.
    _parent = parent

    return m

builtins.__import__ = _import
```

This code chains the built-in `__import__` hook (stored in `_baseimport`). It also tracks the current "parent" module, which is the module that is performing the import operation. Top-level modules won't have a parent.

After a module has been successfully imported, it is added to its parent's dependency list. Note that this code is only interested in file-based modules; built-in extensions are ignored because they can't be reloaded.

This results in a complete set of per-module dependencies for all modules that are imported after this custom import hook has been installed. These dependencies can be easily queried at runtime:

```
def get_dependencies(m):
    """Get the dependency list for the given imported module."""
    return _dependencies.get(m.__name__, None)
```

Reloading Modules

The next step is to build a dependency-aware `reload()` routine.

```
import imp

def _reload(m, visited):
    """Internal module reloading routine."""
    name = m.__name__

    # Start by adding this module to our set of visited modules.  We use
    # this set to avoid running into infinite recursion while walking the
    # module dependency graph.
    visited.add(m)

    # Start by reloading all of our dependencies in reverse order.  Note
    # that we recursively call ourself to perform the nested reloads.
    deps = _dependencies.get(name, None)
    if deps is not None:
        for dep in reversed(deps):
            if dep not in visited:
                _reload(dep, visited)

    # Clear this module's list of dependencies.  Some import statements
    # may have been removed.  We'll rebuild the dependency list as part
    # of the reload operation below.
    try:
        del _dependencies[name]
```

```
    except KeyError:
        pass

    # Because we're triggering a reload and not an import, the module
    # itself won't run through our _import hook.  In order for this
    # module's dependencies (which will pass through the _import hook) to
    # be associated with this module, we need to set our parent pointer
    # beforehand.
    global _parent
    _parent = name

    # Perform the reload operation.
    imp.reload(m)

    # Reset our parent pointer.
    _parent = None

def reload(m):
    """Reload an existing module.

    Any known dependencies of the module will also be reloaded."""
    _reload(m, set())
```

This `reload()` implementation uses recursion to reload all of the requested module's dependencies in reverse order before reloading the module itself. It uses the `visited` set to avoid infinite recursion should individual modules' dependencies cross-reference one another. It also rebuilds the modules' dependency lists from scratch to ensure that they accurately reflect the updated state of the modules.

Custom Reloading Behavior

The reloading module may wish to implement some custom reloading logic, as well. For example, it may be useful to reapply some pre-reloaded state to the reloaded module. To support this, the reloader looks for a module-level function named `__reload__()`. If present, this function is called after a successful reload with a copy of the module's previous (pre-reload) dictionary.

Instead of simply calling `imp.reload()`, the code expands to:

```
# If the module has a __reload__(d) function, we'll call it with a
# copy of the original module's dictionary after it's been reloaded.
callback = getattr(m, '__reload__', None)
if callback is not None:
    d = _deepcopy_module_dict(m)
    imp.reload(m)
    callback(d)
else:
    imp.reload(m)
```

The `_deepcopy_module_dict()` helper routine exists to avoid `deepcopy()`-ing unsupported or unnecessary data.

```
def _deepcopy_module_dict(m):
    """Make a deep copy of a module's dictionary."""
    import copy

    # We can't deepcopy() everything in the module's dictionary because
    # some items, such as '__builtins__', aren't deepcopy()-able.
    # To work around that, we start by making a shallow copy of the
    # dictionary, giving us a way to remove keys before performing the
    # deep copy.
    d = vars(m).copy()
    del d['__builtins__']
```

```
    return copy.deepcopy(d)
```

Monitoring Module Changes

A nice feature of a reloading system is automatic detection of module changes. There are many ways to monitor the file system for source file changes. The approach implemented here uses a background thread and the `stat()` system call to watch each file's last modification time. When an updated source file is detected, its filename is added to a thread-safe queue.

```
import os, sys, time
import queue, threading

_win = (sys.platform == 'win32')

class ModuleMonitor(threading.Thread):
    """Monitor module source file changes"""

    def __init__(self, interval=1):
        threading.Thread.__init__(self)
        self.daemon = True
        self.mtimes = {}
        self.queue = queue.Queue()
        self.interval = interval

    def run(self):
        while True:
            self._scan()
            time.sleep(self.interval)

    def _scan(self):
```

```
# We're only interested in file-based modules (not C extensions).
modules = [m.__file__ for m in sys.modules.values()
        if '__file__' in m.__dict__]

for filename in modules:
    # We're only interested in the source .py files.
    if filename.endswith('.pyc') or filename.endswith('.pyo'):
        filename = filename[:-1]

    # stat() the file.  This might fail if the module is part
    # of a bundle (.egg).  We simply skip those modules because
    # they're not really reloadable anyway.
    try:
        stat = os.stat(filename)
    except OSError:
        continue

    # Check the modification time.  We need to adjust on Windows.
    mtime = stat.st_mtime
    if _win32:
        mtime -= stat.st_ctime

    # Check if we've seen this file before.  We don't need to do
    # anything for new files.
    if filename in self.mtimes:
        # If this file's mtime has changed, queue it for reload.
        if mtime != self.mtimes[filename]:
            self.queue.put(filename)

    # Record this filename's current mtime.
    self.mtimes[filename] = mtime
```

An alternative approach could use a native operation system file monitoring facility, such as the Win32 Directory Change Notification system.

The `Reloader` object polls for source file changes and reloads modules as necessary.

```
import imp
import reloader

class Reloader(object):

    def __init__(self):
        self.monitor = ModuleMonitor()
        self.monitor.start()

    def poll(self):
        filenames = set()
        while not self.monitor.queue.empty():
            try:
                filenames.add(self.monitor.queue.get_nowait())
            except queue.Empty:
                break
        if filenames:
            self._reload(filenames)

    def _reload(self, filenames):
        modules = [m for m in sys.modules.values()
                if getattr(m, '__file__', None) in filenames]

        for mod in modules:
            reloader.reload(mod)
```

In this model, the reloader needs to be polled periodically for it to react to changes. The simplest example would look like this:

```
r = Reloader()
while True:
    r.poll()
    time.sleep(1)
```

The complete source code is on GitHub. The package distribution is available as `reloader` on the Python Package Index.