**Other Projects**

Search eBay for music matching your last.fm profile.
www.exitahead.co.uk

**Other Articles**

Amakode; a media transcoding script for Amarok 1.4. September 2008.

Generating filters for BrutefirDRC. June 2008.

Dependency graphs for Python programs. September 2004.

Using the dvorak keyboard layout to beat acheing forearms. August 2004.

Using spambayes to handle spam email in kmail. July 2004.

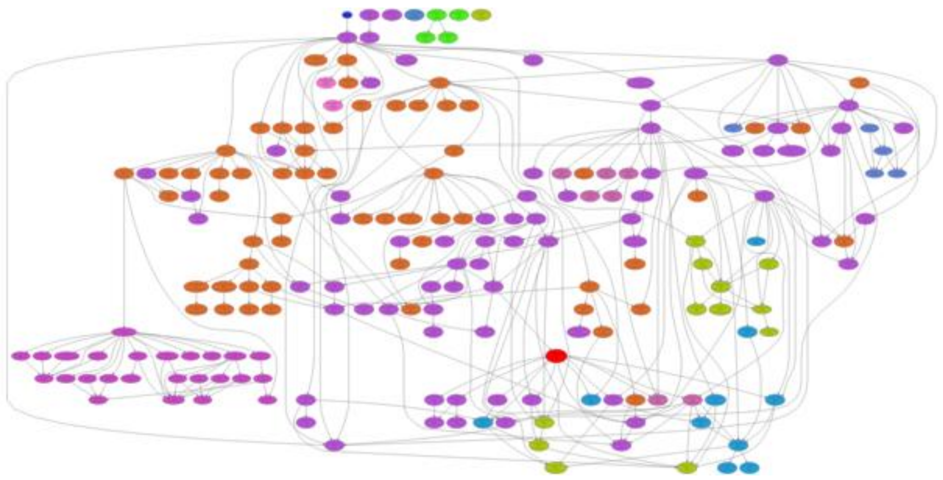
**Contact Me**

toby@tarind.com

# Generating Python Module Dependency Graphs

Controlling physical dependencies is an important part of any software architecture. We noticed a shortage of tools for analysing Python program when we started work on Tintag Explorer, and the tools described here were created as a result.

Below is a shrunken version of the dependency graph from several subsystems of the current development version of Tinytag Explorer; 4.3. The __main__ module is the small dark blue circle at the top. Other circles are other modules. The lines are arrows (although you cant see the arrow-heads in this shrunken version) that indicate that one module imports another. All arrows point down, unless there are cyclic imports.



We have obtained enormous value from studying automatically generated diagrams such as this. It often highlights:

- Ill-advised module imports, as obtrusive diagonal lines across large distances of the diagram.
- Modules that might be in the wrong package, as solitary circles on one color surrounded by those of another color.
- Subsystems that are ideal for reuse in other systems because they have few dependencies. For example, the island of pink on the bottom left of the diagram above has already been reused from another project. [Hi Lawrence.... That's TSG!]
- Close clusters of modules are often a suitable unit for code review.

This diagram was generated from a three-step process:

## Step 1: Compute The Dependency Graph

Python's modulefinder module does all the heavy lifting here. modulefinder use bytecode inspection to find dependencies, and therefore is free from any side-effects that may be caused by importing the modules being studied.

This py2depgraph.py script will write to stdout the dependency graph raw data for the script named in its first parameter.

## Step 2: Format The Dependency Graph

In step 3 we will be passing our data into dot, part of graphviz In step 2 we need to convert the raw dependency data generated in step 1 into the correct format, and apply any presentation logic.

In simple cases you can use depgraph2dot.py as-is. It receives the raw data in standard input, and provides the generated dot file on standard output.

For the best presentation you will need a little programming. Create a subclass of the class defined in this module, override some methods that apply presentation logic, then call the main() method of one of those objects. The following aspects of the presentation can be customised:

1. Change which modules will be used on the diagram. By default it omits a number of very common modules such as sys that would add uninteresting clutter. It also omits packages (but not the modules within the package). The diagram above has been customised to show a group of appliction-specific packages.
2. Pairs of modules that have a specially close relationship. These relationships are drawn with particularly short straight lines. By default any reference to a module whose name starts with an underscore carries this extra weight, because that usually indicates a special relationship (for example, as between `random` and `_random`)
3. Pairs of modules whose relationship should be drawn with an extra long line. This is a good way of separating subsystems - for example the module highlighted in red in the diagram above has extra space above.
4. A color. By default the color is assigned automatically with all modules in one package assigned the same color.
5. A text label, by default the module name.

## Step 3: Generate The Image

You need the dot tool from graphviz. This can generate most image formats. It can also generate postscript ideal for printing. If printing to a black and white printer the --mono switch to depgraph2dot.py will be helpful.

Putting those three steps together gives something like:

```
$ python py2depgraph.py path/to/my/script.py | python depgraph2dot.py | dot -T png -o depgraph.png
$
```

(Created 2004. Updated December 2009 with python 2.6 fixes)