metatools Documentation

Release 0.1

Western X

November 26, 2013

Contents

CHAPTER 1

“Python about Python”

This package contains tools for working with Python source code, or bootstrapping other tools into the Western X execution environment. Applications include:

- creating double-clickable OS X applications which launch Python functions while respecting development environments;
- creating shell executables which launch Python functions while respecting development environments;
- reloading modified code at run-time;
- renaming modules;
- various code quality and/or convention checks or introspections.

CHAPTER 2

Contents

2.1 App Bundles for OS X

2.1.1 Overview

Since our users are often more amenable to the standard GUI interface, we need a way to quickly provide double-clickable apps so that they do not need to use the terminal. While something like Py2App would do the trick, these are unnesesarily heavy-weight for our needs.

Instead, this package directly constructs `.app` packages, with the bare minimum required for OS X to recognize them as applications and know how to launch them; they are essentially stubs for either existing commands or Python functions, but with icons.

Each app is specified by its own YAML file. The following keys are accepted:

`name` The name of the app. Defaults to the base name of the config file. E.g. A YAML file named `toolbox.yml` will default to generating an app named `toolbox`.

`icon` The name of an icon in `$KS_TOOLS/key_base/2d/icons`, or an absolute path. Defaults to the value of `name`.

`entrypoint` Either `"package.module"` or `"package.module:function"` specifying which module to import/run

`command` A command to run as if from a shell. Defaults to the value of `name` if `entrypoint` is also not specified. Cannot be specified along with `entrypoint`.

As you can see, given the defaults above, it is possible to fully describe an app in this manner with a completely empty YAML file!

2.1.2 Building Apps

Apps are build by running the main method of the `metatools.apps` module. It takes any number of YAML files as arguments, with the last argument being the directory to save the apps into. For example:

```
python -m metatools.apps *.yml ./
```

will build all the YAML files in the current directory into apps also in the current directory.

2.2 Entrypoint Scripts

2.2.1 API Reference

`metatools.entrypoints.build`

`metatools.entrypoints.build.`**`dedent`**`(`*docstring*`)`

`metatools.entrypoints.search`

This module deals with finding absolute paths to executables or apps to run. It is imperative that we return absolute paths so that the executables will be able to bootstrap a proper development environment.

Apps are searched for within `key_base/applications` in all tools listed in `KS_PYTHON_SITES` and in `KS_TOOLS`. Executables are searched for within `PATH`.

`metatools.entrypoints.search.`**`get_executable_path`**`(`*name*`)`

Find an executable on the current $PATH.

Parameters **name** (*str*) – The name of the executable to find.

Throws ValueError When it cannot find an executable.

```
>>> get_executable_path('toolbox')
<snip $KS_TOOLS>/key_base/bin/generated/toolbox
```

`metatools.entrypoints.search.`**`get_app_or_executable_cmd`**`(`*app_name, exec_name=None*`)`

Find an OS X app (if on a Mac), and fall back to an executable.

Parameters

- **app_name** (*str*) – The name of the OS X application to find (without `.app`).
- **exec_name** (*str*) – The name of the executable to find; defaults to `app_name`.

Throws ValueError When it cannot find an app or executable.

Returns A list that is directly usable in `subprocess.Popen`, and which anything appended to will be treated as an argument.

On a Mac when there is an application:

```
>>> get_app_or_executable_cmd('qb-reelsmart', 'qb_reelsmart')
['open', '<snip $KS_TOOLS>/key_base/qb-reelsmart.app', '--args']
```

On Linux, or when there is no application:

```
>>> get_app_or_executable_cmd('qb-reelsmart', 'qb_reelsmart')
['<snip $KS_TOOLS>/key_base/bin/generated/qb_reelsmart']
```

2.3 Configuration and User Preferences

Tools for easy persistance of configuration and user preferences for tools.

Configurations are split into sections and keys; sections are conceptually for each tool (or other grouping of settings), and keys are for individual settings within that tool/section. In the current implementation, individual sections are saved within YAML files as a mapping.

Basic usage:

```
# Creating the config object; give it a unique name. Slashes will be
# used directly to specify sub-directories.
config = metatools.config.Config('your_company/' + __name__)

# Use a value within the config; treat it like a dict.
dialog = setup_gui(width=config.get('width', 800))

# Save the values later.
config['width'] = dialog.width()
config.save() # Only if there were changes to the values.
```

Quick use:

```
width = metatools.config.get('your_company/' + __name__, 'width')
metatools.config.set('your_company/' + __name__, 'width', width)
```

class `metatools.config.`**`Config`**`(`*name*`)`

Mapping which persists to disk via YAML serialization.

Use like a dictionary.

`revert``()`

Revert to saved state.

`save``(`*force=False*`)`

Persist the current contents.

Parameters **force** (*bool*) – Always write, even if there were no changes.

`delete``()`

Clear and delete; same as clear() and save().

`update``(`**args*, ***kwargs*`)`

`metatools.config.`**`get`**`(`*section*, *name*, **args*`)`

`metatools.config.`**`set`**`(`*section*, *name*, *value*`)`

`metatools.config.`**`main`**`()`

2.4 Lint

Todo

Write this.

2.5 Monkeypatching

`metatools.monkeypatch.`**`patch`**`(`*to_patch*, *name=None*, *must_exist=True*, *max_version=None*`)`

Monkey patching decorator.

Parameters

- **to_patch** – The object to patch.
- **name** (*str*) – The attribute of the object to patch; defaults to name of the patch function.

- **must_exist** (*bool*) – Must the original exist for the patch to be applied?
- **max_version** (*tuple*) – The maximum Python version to apply this patch to.

Returns A decorator which takes a function and applies the patch, returning the patched version.

Usage:

```
# Patch os.listdir to return all lowercase.
@patch(os)
def listdir(original, path):
    return [x.lower() for x in original(path)]

# It is patched in place.
os.listdir('.')

# The new function still refers to the original.
listdir('.')

# Patch chflags to do nothing in Python up to 2.6.
@patch(os, 'chflags', max_version=(2, 6))
def patched_chflags(func, *args, **kwargs):
    pass
```

2.6 Imports and Modules

2.6.1 Discovery of Dependencies

`metatools.imports.discovery.`**`get_toplevel_imports`**(*module*)

Get the imports at the top-level of the given Python module.

Parameters **module** – An actual module; not the name.

Returns list The absolute names of everything imported,

`metatools.imports.discovery.`**`parse_imports`**(*source*, *package=None*, *module=None*, *path=None*, *toplevel=True*)

Get the imports at the top-level of the given Python module.

Parameters

- **source** (*str*) – Python source code.
- **package** (*str*) – The `__package__` this source is from.
- **module** (*str*) – The `__name__` this source is from.
- **toplevel** (*bool*) – Walk the full AST, or only look at the top-level?

Returns list The names of everything imported; absolute if package and module are provided.

`metatools.imports.discovery.`**`path_is_in_directories`**(*path*, *directories*)

Is the given path within the given directory?

Parameters

- **path** (*str*) – The path to test.
- **directory** (*str*) – The directory to test if the path is in.

Returns bool

2.6.2 Reloading Code in Production

Automatically reloading modules when their source has been updated.

This also allows for some state retention in reloaded modules, for modules to specify dependencies that must be checked for being outdated as well, and to unload those dependencies on reload.

There are two ways that we use to track if something should be reloaded, the last modification time of a file, and the last time we reloaded a module.

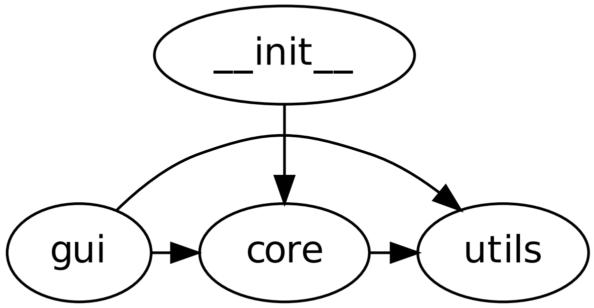
The modification time is to determine if a module has actually changed, and if it is newer than a previously seen time, then that module will be reloaded.

Every module reload will store the time at which it reloads (determined at the start of an autoreload cycle). Then, in another autoreload cycle, another module can see when it's dependencies changes by comparing its reload time with the dependency's. If a dependency was reloaded, then reload ourselves as well.

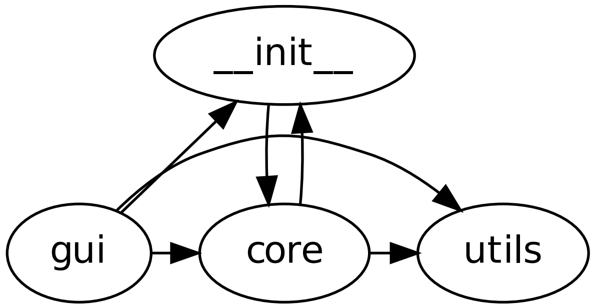
This is not perfect, and it is known to cause numerous issues (e.g. circular imports cause some strange problems), however we have found that the problems it brings up are minor in comparison to the speed boost it tends to give us while in active development.

A better algorithm would construct a full module graph (it would not be acyclic), and iteratively expand the region that must be reloaded. Then it would linearize the dependencies and reload everything in a big chain.

The tricky part is since *module discovery <discovery>* does not reveal the actual intensions of the code, e.g.:



but all the dependancies that it is actually capable of, e.g.:



metatools.imports.reload.**is_outdated**(*module*, *recursive=True*)

metatools.imports.reload.**reload**(*module*, *_time=None*)

metatools.imports.reload.**autoreload**(*module*, *visited=None*, *force_self=None*, *_depth=0*, *_time=None*)

2.6.3 Rewriting Imports

class metatools.imports.rewrite.**Rewriter**(*mapping*, *module_name*)

add_substitution(*source*)

convert_identifier(*name*)

convert_module(*name*)

direct_import(*m*)

import_from(*m*)

make_relative(*target*)

split_as_block(*block*)

usage(*m*)

metatools.imports.rewrite.**diff_texts**(*a*, *b*, *filename*)
Return a unified diff of two strings.

metatools.imports.rewrite.**main**()

metatools.imports.rewrite.**module_name_for_path**(*path*)

metatools.imports.rewrite.**resolve_relative**(*relative*, *module*)

metatools.imports.rewrite.**rewrite**(*source*, *mapping*, *module_name=None*, *non_source=False*)

2.6.4 Utilities

metatools.imports.utils.**get_name_for_path**(*path*)

metatools.imports.utils.**get_source_path**(*module*) `metatools.imports.utils.`**`resolve_relative_name`**(*package*, *module*, *relative*)

Convert a relative import path into an absolute one.

Parameters

- **package** (*str*) – The `__package__` that we are in.
- **module** (*str*) – The `__name__` of the module we are in.
- **relative** (*str*) – The module name to resolve.

Absolute names are passed through untouched.

2.7 Deprecating Code

exception `metatools.deprecate.`**`CallingDeprecatedWarning`**

exception `metatools.deprecate.`**`AttributeRenamedWarning`**

exception `metatools.deprecate.`**`FunctionRenamedWarning`**

exception `metatools.deprecate.`**`ModuleRenamedWarning`**

`metatools.deprecate.`**`renamed_attr`**

Proxy for renamed attributes (or methods) on classes.

Getting and setting values will be redirected to the provided name, and warnings will be issues every time.

E.g.:

```
>>> class Example(object):
...
...     new_value = 'something'
...     old_value = renamed_attr('new_value')
...
...     def new_func(self, a, b):
...         return a + b
...
...     old_func = renamed_attr('new_func')

>>> e = Example()

>>> e.old_value = 'else'
# AttributeRenamedWarning: Example.old_value renamed to new_value

>>> e.old_func(1, 2)
# AttributeRenamedWarning: Example.old_func renamed to new_func
3
```

`metatools.deprecate.`**`renamed_func`**(*func*, *name=None*, *module=None*)

Proxy for renamed functions.

Parameters

- **func** – The function to actually call.
- **name** (*str*) – The name that this used to be called; for warnings.
- **module** (*str*) – The module that this used to be in; for warnings.

Returns A function which calls the original, and omits a warning.

E.g.:

```
>>> def new(a, b):
...     return a + b

>>> old = renamed_func(new, 'old', __name__)

>>> old(1, 2)
# FunctionRenamedWarning: example.old renamed to example.new
3
```

`metatools.deprecate.`**`module_renamed`**`(`*new_name*`)`

Replace the current module with the one found at the given name.

Issues a `ModuleRenamedWarning`.

For example,

`new.py`:

```
>>> def func():
...     print "Hello from %s!" % __name__
```

`old.py`:

```
>>> from metatools.deprecate import module_renamed
>>> module_renamed('new')
```

`use.py`:

```
>>> from old import func
# ModuleRenamedWarning: old was renamed to new
>>> func()
Hello from new!
```

`metatools.deprecate.`**`deprecate`**`(`*func*`)`

Wrap a function so that it will issue a deprecation warning.

```
>>> @deprecate
... def old_func():
...     print "Hello!"
...
>>> old_func()
# CallingDeprecatedWarning: example.old_func has been deprecated
Hello!
```

Python Module Index