entrypoint2 Documentation

Release 0.0.2

ponty

April 28, 2011

CONTENTS

entrypoint2 is an easy to use command-line interface for python modules, fork of entrypoint

Date: April 28, 2011

Contents:

entrypoint2 is an easy to use command-line interface for python modules, fork of entrypoint

Links:

- home
- html documentation
- pdf documentation

CHAPTER

ONE

BACKGROUND

There are tons of command-line handling modules but none of them can generate a CLI interface for a very simple function like this without duplicating the existing code and without making the function from other modules unusable:

```
def add(one, two=4, three=False):
        ''' description
        one: description
        two: description
        three: description
        '''
```

Best solution I could find is entrypoint, but there is no link to development site, so I forked the project. The only big disadvantage of entrypoint: it destroys the function signature, therefore the function can not be called from other modules.

CHAPTER

TWO

GOALS

- the decorated function should have the same behavior as without the entrypoint2 decorator
- generate CLI parameters from function signature
- generate CLI documentation from python documentation
- boolean parameters should be toggle flags
- generate short flags from long flags: `--long -> -l`
- automatic –version flag

CHAPTER

THREE

SIMILAR PROJECTS

- entrypoint
- plac
- baker
- argh
- opster

CHAPTER

FOUR

FEATURES

Additional features over original entrypoint:

- function signature is preserved so it can be called both from command-line and external module
- function name, doc and module are preserved so it can be used with sphinx autodoc
- sphinx autodoc documentation style is supported: `:param x:  this is x`
- automatic `--version` flag, which prints version variable from the current module (`__version__`, `VERSION`, ..)
- automatic `--debug` flag, which turns on logging
- short flags are generated automatically (e.g. `--parameter` -> `-p`)
- unit tests

Known problems:

- Python 3 is not supported
- there are more decorators in the module inherited from original entrypoint, but only @entrypoint is tested.
- Autocompletion is not supported

CHAPTER

FIVE

BASIC USAGE

Example:

```
from entrypoint2 import entrypoint

__version__ = '3.2'

@entrypoint
def add(one, two=4, three=False):
    ''' This function adds three numbers.

    one: first number to add
    two: second number to add
    '''
```

Generated help:

```
$ python -m entrypoint2.examples.hello --help
usage: hello.py [-h] [-t TWO] [--three] [--version] [--debug] one

This function adds two number.

positional arguments:
  one                 first number to add

optional arguments:
  -h, --help          show this help message and exit
  -t TWO, --two TWO   second number to add
  --three
  --version           show program's version number and exit
  --debug             set logging level to DEBUG
```

Printing version:

```
$ python -m entrypoint2.examples.hello --version
3.2
```

CHAPTER

SIX

INSTALLATION

6.1 General

- install setuptools or pip
- install the program:

if you have setuptools installed:

```
# as root
easy_install entrypoint2
```

if you have pip installed:

```
# as root
pip install entrypoint2
```

6.2 Ubuntu

```
sudo apt-get install python-setuptools
sudo easy_install entrypoint2
```

6.3 Uninstall

```
# as root
pip uninstall entrypoint2
```

CHAPTER

SEVEN

ORIGINAL ENTRYPOINT DOCUMENTATION

Source: http://pypi.python.org/pypi/entrypoint/

This is a decorator library that helps one to write small scripts in Python.

There are three main features that it provides:

- Automatically running a function when a script is called directly, but not when it is included as a module.
- Automatically generating argument parsers from function signatures and docstrings
- Automatically opening and closing files (with a codec or as binary) when a function is called and returns.

The raison d'être of this library is to be convenient, and it makes some sacrifices in other areas. Notably there are some stringent conditions on the order of application of decorators.

For further information, see the explanations below, and the docstrings of the functions. To report errors, request features, or submit patches, please email conrad.irwin@gmail.com.

I've tried to comment-out sections of the documentation that are less relevant to people trying to simply use the library. They contain background information or extra interfaces for re-using components.

1. **Automatically running a function when the library has been called directly.** This is conventionally done in Python using:

```
>>> def main():
...     pass

>>> if __name__ == '__main__':
...     main();
```

And can be re-written to

```
>>> @autorun
... def main():
...     pass
```

Warning: This will not work as expected unless it is the "outermost" decorator, i.e. the decorator that is listed first in the file and applied last to the function. If you only have one decorator, that should be fine - and that's why the library also provides @entrypoint, @entrywithfile and @runwithfile (see table below) so that you only need one decorator.

Specifically, this decorator will call the function as part of the process of decorating. Thus any decorators that are applied after this one will not have been applied in the case the function is called later.

autorun can also be used as a standalone function, which is necessary if you would like to use this functionality as part of another library. In such a case you need to pass it a second parameter indicating how far many levels above in the stack frame within autorun you would expect for __name__ to equal '__main__'.

```
>>> def autorunwithone(func):
...     autorun(lambda: func(1), 2)
>>> @autorunwithone
... def puts(y):
...     print y
```

2. Automatically opening and closing files with the appropriate encoding.

   This is conventionally done using:

   ```
   >>> from __future__ import with_statement
   >>> def main(filename):
   ...     with codecs.open(filename, 'r', 'utf-8') as openfile:
   ...         pass
   ```

   And can be re-written to

   ```
   >>> @withfile('r')
   ... def main(openfile):
   ...     pass
   ```

   It is possible to pass a codec's name as the first positional argument (or as __encoding) to @withfile. The default encoding is stored in entrypoint.ENCODING and is set to 'utf-8'.

   ```
   >>> @withfile('utf-16', 'w')
   ... def main(openfile):
   ...     pass
   ```

   Or to open files in "binary" mode, with no codec, just suffix the spec with a 'b':

   ```
   >>> @withfile('rb', 'a')
   ... def main(binaryfile, logfile):
   ...     print >>logfile, process(binaryfile.read())
   ```

   WARNING: Default arguments to functions are opened and closed on each entry to that function, when a function will be called more than once used 'a' instead of 'w' so that later calls don't overwrite the contents.

   ```
   >>> @withfile('r', 'a')
   ... def main(readfile, log='/tmp/python.log'):
   ...     log.write("Reading %s" % readfile.name)
   ```

   For clarity, it is possible to give keyword arguments to @withfile, and it is necessary to do so if you wish to open all the arguments provided to the function's *args or **kwargs:

   ```
   >>> @withfile('w', args='r', stderr='a')
   ... def main(catfile, *args, **kwargs):
   ...     if args:
   ...         catfile.write("\n".join(arg.read() for arg in args))
   ...     elif 'stderr' in kwargs:
   ...         print >>kwargs['stderr'], "Nothing to cat"
   ```

   Finally, following the convention of many command line tools, the special filename '-' is used to refer to sys.stdin for reading, and sys.stdout for writing and appending. Again this can be a default parameter or passed in by the caller:

```
>>> @withfile('r', 'w')
... def main(input, output='-'):
...     pass
```

WARNING: The files are opened on entry to the function, not when you need them, if you open a file for writing, it will be created on disk, even if you don't write anything to it.

3. **Automatically parsing command-line arguments from a function's signature,** and, if possible, from its doc-string.

   Internally, this uses the argparse module, but removes the tedious syntax needed to get the most simple arguments parsed.

   At its most basic, it simply converts a function that takes several positional arguments (**kwargs is not supported) into a function that takes an optional array of arguments, and defaults to sys.argv[1:]

```
>>> @acceptsargv
... def main(arg1, arg2):
...     pass
...
... main()
... main(sys.argv[1:])
... main(['arg1', 'arg2'])
```

   This can be coupled with the other magic above, so that the function is called automatically when it is defined:

```
>>> sys.argv[1:] = ['arg1', 'arg2']
>>> @entrypoint
... def main(arg1, arg2)
...     pass
```

   The argument parser will abort the program if the arguments don't match, and print a usage message. More detail can be found by passing -h or –help at the command line as is normal.

```
>>> @entrypoint
... def main(arg1, arg2):
...     pass
```

   usage: test.py [-h] arg1 arg2 : error: too few arguments

   In addition to compulsary, positional, arguments as demonstrate above it is possible to add flag arguments. Flag arguments are signified by providing a default value for the parameter, of the same type as you wish the user to input. Positional arguments, and flags with a default value of None are always decoded as unicode strings. If the type conversion fails, it is presented to the user as an error.

```
>>> @entrypoint
... def main(filename, priority=1):
...     assert isinstance(priority, int)
```

   usage: [-h] [–priority PRIORITY] filename

   If the default value is True or False, the flag will be treated as a toggle to flip that value:

```
>>> @entrypoint
... def main(filename, verbose=False):
...     if verbose:
...         print filename
```

   usage: [-h] [–verbose] filename

It is also possible to use the *args of a function:

```
>>> @entrypoint
... def main(output, *input):
...     print ", ".join(filenames)
```

usage: [-h] output [input [input ...]]

In addition to being able to parse the arguments automatically, @acceptargv can also be used to provide user-facing documentation in the same manner as argparse. It does this by parsing the function's doc string in the following ways:

```
>>> def main(filename, flag=True, verbosity=3):
...     """
...         Introductory paragraph.
...
...         Description and clarification of arguments.
...
...         Epilogue
...
...         ----
...
...         Internal documentation
...     """
...     pass
```

All parts are optional. The introductory paragraph and the epilogue are shown before and after the summary of arguments generated by argparse. The internal documentation (below the —-) is not displayed at all:

```
<argument> = <clarification>:<description>
<clarification> = [-<letter>[,] ] --<flagname> [=<varname>]
                = <argname> [/<displayname>]
```

The description can span multiple lines, and will be re-formed when displayed.

In the first case, the -<letter> gives a one-letter/number abbreviation for setting the flag:

```
>>> def main(flag=True):
...     """
...         -f --flag: Set the flag
...     """
```

```
<argname>, <flagname>, <varname>, and <displayname> are limited to
[a-zA-Z][a-zA-Z0-9_-]*
```

The flagname and the argname should match the actual name used in the function argument definition, while the displayname and varname are simply for displaying to the user.

Finally, any function that is wrapped in this manner can throw an entrypoint.UsageError, the first parameter of which will be displayed to the user as an error.

Several combinations are available as pre-defined decorators:

	Run Automatically	Signature Parser	Open Files
@autorun	X		
@entrypoint	X	X	

```
@entrywithfile     X           X           X*
@runwithfile       X                       X
@withfile                                  X
@withuserfile                              X*
@acceptargv                    X
```

- Denotes that FileUsageErrors will be thrown instead of IOErrors to provide more user-friendly error reporting

A set of tests can be run by calling "python test.py"

CHAPTER

EIGHT

USAGE

8.1 CLI

Example program which adds 2 numbers:

```
from entrypoint2 import entrypoint
import logging

__version__ = '3.2'

@entrypoint
def add(one, two=4, three=False):
    ''' This function adds two number.

    one: first number to add
    :param two: second number to add
    :rtype: int
    '''
    sum = str(int(one) + int(two))

    logging.debug('logging sum from hello.py:' + sum)
    print 'printing sum from hello.py:', sum

    return sum
```

Printing help with `--help`:

```
$ python -m entrypoint2.examples.hello --help
usage: hello.py [-h] [-t TWO] [--three] [--debug] [--version] one

This function adds two number.

positional arguments:
  one                 first number to add

optional arguments:
  -h, --help          show this help message and exit
  -t TWO, --two TWO   second number to add
  --three
  --debug             set logging level to DEBUG
  --version           show program's version number and exit
```

Printing version with `--version`:

```
$ python -m entrypoint2.examples.hello --version
3.2
```

Printing sum of two number by the program:

```
$ python -m entrypoint2.examples.hello 3 --two 2
printing sum from hello.py: 5
```

The same but logging is activated:

```
$ python -m entrypoint2.examples.hello 3 -t 2 --debug
DEBUG:root:logging sum from hello.py:5
printing sum from hello.py: 5
```

8.2 calling from other module

Example program which calls the adding function in previos module:

```
import hello
from entrypoint2 import entrypoint
import logging

__version__='5.2'

@entrypoint
def f():
    ''' calls hello
    '''
    sum=hello.add(7,2)

    logging.debug('logging sum from caller.py:' + sum)
    print 'printing sum from caller.py:', sum
```

Calling without logging:

```
$ python -m entrypoint2.examples.caller
printing sum from hello.py: 9
printing sum from caller.py: 9
```

Calling with logging:

```
$ python -m entrypoint2.examples.caller --debug
DEBUG:root:logging sum from hello.py:9
DEBUG:root:logging sum from caller.py:9
printing sum from hello.py: 9
printing sum from caller.py: 9
```

8.3 Example API

API doc of example program generated by autodoc:

entrypoint2.examples.hello.**add**(*one*, *two=4*, *three=False*)

This function adds two number.

one: first number to add :param two: second number to add :rtype: int

CHAPTER

NINE

API

entrypoint2.**entrypoint**(*func*)

A decorator for your main() function.

Really a combination of @autorun and @acceptargv, so will run the function if __name__ == '__main__' with arguments extricated from argparse.

As with @acceptargv, this must either be the innermost decorator, or separated only by "well-behaved" decorators that preserve the __doc__ attribute AND the function signature.

As with @autorun, this must be theoutermost decorator, as any decorators further out will not be applied to the function until after it is run.

exception entrypoint2.**UsageError**(*message*)

When a function wrapped with @acceptargv or @entrypoint raises this exception, the message will be printed to the user implying that it was their fault that things have gone horribly wrong.

CHAPTER

TEN

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

e

INDEX