Python Loger Indenter and Helper Documentation

Release 0.9

Dan Strohl

March 28, 2016

Contents

This module helps by allowing you to indent the lines in your log entries to make it easier to read when you are dealing with complex loops and calls (especially with debug level logging)

This subclasses a standard `logging.LoggerAdapter` and allows you to control the indent level of your message.

Contents:

CHAPTER 1

Installation and initial setup

To install the indented log helper, first get it using `pip`:

```
pip install python_log_indenter
```

Since this is, in effect, a logging adapter, you simply wrap your logger in the IndentedLoggerAdapter class:

```
from python_log_indenter import IndentedLoggerAdapter
import logging

log = IndentedLoggerAdapter(logging.get_logger(__name__))
```

You can now use it as you would any other log:

```
log.debug('this is a debug log entry')
```

see the Basic usage page for examples of how to use this.

CHAPTER 2

Basic usage

2.1 Loading

This subclasses a standard `logging.LoggerAdapter`, so you simply wrap your standard logger object in this and use it as normal. The `IndentedLoggerAdapter` has the same methods as the normal `logging.Logger`, so you dont have to change any existing code.

Example:

```
from python_log_indenter import IndentedLoggerAdapter
import logging

logging.basicConfig(level=logging.DEBUG)
log = IndentedLoggerAdapter(logging.get_logger(__name__))
```

You can also chance the default number of spaces used for indenting, as well as chance the character used (see Advanced usage for more information)

2.2 Changing the indent level

Once you have loaded the IndentedLoggerAdapter, you can change the level of the indents as you go using the .add / .sub methods:

```
>>> log.info('test1')
>>> log.add()
>>> log.info('test2')
>>> log.sub()
>>> log.info('test3')
```

Will result in a log looking like:

```
INFO:root:test1
INFO:root:    test2
INFO:root:test3
```

You can also add or subtract multiple levels by passing an int to `IndentedLoggerAdapter.add()` or `IndentedLoggerAdapter.sub()`:

```
>>> log.info('test1')
>>> log.add(3)
>>> log.info('test2')
```

```
>>> log.sub(2)
>>> log.info('test3')

# returning
INFO:root:test1
INFO:root:            test2
INFO:root:    test3
```

In addition, the `IndentedLoggerAdapter.add()` and `IndentedLoggerAdapter.sub()`:: are chainable (along with several of the other methods). This means you can clean up your code to look like:

```
>>> log.info('test1')
>>> log.add().info('test2').sub()
>>> log.info('test3')
```

2.2.1 Other Features

There are several other features included in this library, these are documented in the Advanced usage section. These including:

Push/Pop: The ability to push or pop indent levels from a FILO queue.

Memories: The ability to store indent levels into a named memory location.

Formatable as fields: The ability to add the indent as a field to the `logging.LogRecord` so that it can be included or not based on the format string and the handler.

Shortcuts: Shortcut methods for many of the fields for quicker usage.

CHAPTER 3

Advanced usage

the py:class:*IndentedLoggerAdapter* has several other methods and usages for advanced usage.

3.1 Push / Pop

`IndentedLoggerAdapter.push()` and `IndentedLoggerAdapter.pop()` allow you to add an indent level to a first in last out queue (FILO so that you can save an indent level and go back to it, even if you dont remember what it was.

Example:

```
>>> log.info('test1').push().add()
>>> log.info('test2')
>>> log.add().info('test3')
>>> log.pop().info('test4')

# returning

INFO:root:test1
INFO:root:    test2
INFO:root:        test3
INFO:root:test4
```

This can be helpfull if you are changing indents across methods or functions:

```
def test1():
log.info('entering test function').add()

# do something ...

log.push()  # save the location
log.add().debug('entering the loop')
for i in range(3):
    sub_test(i)

log.pop()    # getting the saved location
log.debug('leaving function')

def sub_test(cnt):
    log.debug('sub_test_loop %d', cnt)

    # do some loopy thing ...
```

```
    log.a().debug('doing loopy stuff').s()

# returns

INFO:root:entering test function
DEBUG:root:    entering the loop
DEBUG:root:        sub_test_loop 0
DEBUG:root:            doing loopy stuff
DEBUG:root:        sub_test_loop 1
DEBUG:root:            doing loopy stuff
DEBUG:root:        sub_test_loop 2
DEBUG:root:            doing loopy stuff
DEBUG:root:    leaving the function
```

Especially if you forget to return to the same level as before. For example, if we run the above test1() function three times we would see:

```
INFO:root:entering test function
DEBUG:root:    entering the loop
DEBUG:root:        sub_test_loop 0
DEBUG:root:            doing loopy stuff
DEBUG:root:        sub_test_loop 1
DEBUG:root:            doing loopy stuff
DEBUG:root:        sub_test_loop 2
DEBUG:root:            doing loopy stuff
DEBUG:root:    leaving the function
INFO:root:    entering test function
DEBUG:root:        entering the loop
DEBUG:root:            sub_test_loop 0
DEBUG:root:                doing loopy stuff
DEBUG:root:            sub_test_loop 1
DEBUG:root:                doing loopy stuff
DEBUG:root:            sub_test_loop 2
DEBUG:root:                doing loopy stuff
DEBUG:root:        leaving the function
INFO:root:        entering test function
DEBUG:root:            entering the loop
DEBUG:root:                sub_test_loop 0
DEBUG:root:                    doing loopy stuff
DEBUG:root:                sub_test_loop 1
DEBUG:root:                    doing loopy stuff
DEBUG:root:                sub_test_loop 2
DEBUG:root:                    doing loopy stuff
DEBUG:root:            leaving the function
```

if we had used `IndentedLoggerAdapter.push()` and `IndentedLoggerAdapter.pop()` at the beginning and end of the method, we would have cleared out the building indent.

3.1.1 Push / Pop by name

You can also push and pop by name, this allows you to set a name while pushing an indent level, then return to that point in the queue without having to do multiple pop's.

For example:

```
def test1():
log.push('test1_function')
log.info('entering test function').add()
```

```
# do something ...

log.add().debug('entering the loop')
log.push()
for i in range(3):
    sub_test(i)

log.debug('leaving function')

# This pops TWO levels from the queue, the first one (Just above the "for / in") and returns to the f
log.pop('test1_function')

def sub_test(cnt):
    log.debug('sub_test_loop %d', cnt)

    # do some loopy thing ...
    log.a().debug('doing loopy stuff').s()

# returns

INFO:root:entering test function
DEBUG:root:    entering the loop
DEBUG:root:        sub_test_loop 0
DEBUG:root:            doing loopy stuff
DEBUG:root:        sub_test_loop 1
DEBUG:root:            doing loopy stuff
DEBUG:root:        sub_test_loop 2
DEBUG:root:            doing loopy stuff
DEBUG:root:    leaving the function
```

In addition, you can pass the indent level to the .push() (without changing the current level), and you can pass the number of levels to go back to the .pop():

```
>>> log.info('test1').push(2).add()
>>> log.info('test2')
>>> log.add().info('test3').push()
>>> log.info('test4')
>>> log.pop().info('test5')

# returning

INFO:root:test1
INFO:root:        test2
INFO:root:            test3
INFO:root:            test4
INFO:root:test4
```

3.2 Memories

`IndentedLoggerAdapter.mem_save()`, `IndentedLoggerAdapter.mem()`, and `IndentedLoggerAdapter.mem_clear()` You also can store indent levels using named storage locations, this allows you to setup indent levels for specific things and recall them as needed.:

```
>>> log.mem_save('level1',1)
>>> log.mem_save('level2',2)
>>> log.mem_save('level3',3)
```

```
>>> log.info('test0')
>>> log.mem('level1').info('test1')
>>> log.info('test2')
>>> log.mem('level2').info('test3')
>>> log.mem('level3').info('test4')
>>> log.mem('level1').info('test5')

# returning

INFO:root:test0
INFO:root:    test1
INFO:root:    test2
INFO:root:        test3
INFO:root:            test4
INFO:root:test5
```

If you do not pass an indent level to .mem_save() it will save the current level.

3.3 Formatting

By default the library will add the indent to the beginning of the message string, however if you want more control over the formatting of the log string, you can change the behaivior to set the indent_str as a `logging.LogRecord` property, which can then be access by format strings set in the logging configuration.

This allows you to use the indenting for console logging, but not for log files (or any other mix you want). In addition, the indent_level is available as well if you want to pass that into the formatting string.

These are available using the "indent_str" and "indent_level" keywords in the formatting string.

As an example of a useless format:

```
logging.basicConfig(format='%(name)-8s: %(levelname)-8s : level %(indent_level) : indent <%(indent_st
log = IndentedLoggerAdapter(logging.getLogger(), spaces=1, indent_char='.', auto_add=False)

log.info('test1')
log.add(3)
log.info('test2')
log.sub(2)
log.info('test3')

# returning
root    : INFO     : level 0 : indent <> : test 1
root    : INFO     : level 3 : indent <            > : test 1
root    : INFO     : level 1 : indent <    > : test 1
```

for better examples, see the logging cookbook on the `logging`

3.4 Shortcuts

Shortcut methods have also been defined to assist in making these faster to enter (not that the names are very long to begin with).

Method	Shortcut
.indent_level()	.i()
.add()	.a()
.sub()	.s()
.pop()	.po()
.push()	.pu()
.mem()	.m()
.mem_save()	.ms()
.mem_clear()	.mc()

Also, you can access memory location using dictionary methods, for example:

```
>>> log['level1'] = 1
>>> log['level2'] = 2
>>> log['level3'] = 3
>>> log.info('test0')
>>> log['level1'].info('test1')
>>> log.info('test2')
>>> log['level2'].info('test3')
>>> log['level3'].info('test4')
>>> log['level1'].info('test5')

# returning

INFO:root:test0
INFO:root:    test1
INFO:root:    test2
INFO:root:        test3
INFO:root:            test4
INFO:root:test5
```

3.5 Changing indent size and indent character

When loading the `IndentedLoggerAdapter` you can choose to set the size of the indent and the character used to create the indent.

For example:

```
logging.basicConfig(level=logging.DEBUG)
log = IndentedLoggerAdapter(logging.getLogger(), spaces=1, indent_char='.')

log.a().info('test 1')
log.s().error('test 2')
log.a(3).debug('test 3')
log.push().warning('test 4')
log.a(1).critical('test 5')
log.pop().critical('test 6')

INFO:root:.test 1
ERROR:root:test 2
DEBUG:root:...test 3
WARNING:root:...test 4
CRITICAL:root:....test 5
CRITICAL:root:...test 6
```

See the IndentedLoggerAdapter API section for information on the api for specific parameters.

CHAPTER 4

IndentedLoggerAdapter API

API Reference for the IndentedLoggerAdapter

This will allow you to turn logs that would normally look like this:

```
root    DEBUG   Loading system
root    DEBUG   Checking for the right record
root    DEBUG   Checking record 1
root    DEBUG   Checking name
root    DEBUG   Not the right record
root    DEBUG   Checking record 2
root    DEBUG   checking name
root    DEBUG   Name checks, checking phone number
root    DEBUG   Phone number checks, VALID record!
root    DEBUG   Returning record 2
```

Into something like this:

```
root    DEBUG   Loading system
root    DEBUG       Checking for the right record
root    DEBUG           Checking record 1
root    DEBUG               Checking name
root    DEBUG                   Not the right record
root    DEBUG           Checking record 2
root    DEBUG               checking name
root    DEBUG                   Name checks, checking phone number
root    DEBUG               Phone number checks, VALID record!
root    DEBUG           Returning record 2
```

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Note: Please feel feel free to make bug reports at https://github.com/dstrohl/Python_log_indenter/issues as well as forking the repo and issuing pull requests!