Logging All Versions

Examples

Introduction to Python Logging

This module defines functions and classes which implement a flexible event logging system for applications and libraries.

The key benefit of having the logging API provided by a standard library module is that all Python modules can participate in logging, so your application log can include your own messages integrated with messages from third-party modules.

So, lets start:

Example Configuration Directly in Code

```
import logging

logger = logging.getLogger()
handler = logging.StreamHandler()
formatter = logging.Formatter(
        '%(asctime)s %(name)-12s %(levelname)-8s %(message)s')
handler.setFormatter(formatter)
logger.addHandler(handler)
logger.setLevel(logging.DEBUG)

logger.debug('this is a %s test', 'debug')
```

Output example:

```
2016-07-26 18:53:55,332 root         DEBUG    this is a debug test
```

Example Configuration via an INI File

Assuming the file is named logging_config.ini. More details for the file format are in the logging configuration section of the logging tutorial .

```
[loggers]
keys=root

[handlers]
keys=stream_handler

[formatters]
keys=formatter

[logger_root]
level=DEBUG
handlers=stream_handler

[handler_stream_handler]
class=StreamHandler
level=DEBUG
formatter=formatter
args=(sys.stderr,)

[formatter_formatter]
format=%(asctime)s %(name)-12s %(levelname)-8s %(message)s
```

Then use logging.config.fileConfig() in the code:

```
import logging
from logging.config import fileConfig

fileConfig('logging_config.ini')
logger = logging.getLogger()
logger.debug('often makes a very good meal of %s', 'visiting tourists')
```

Example Configuration via a Dictionary

As of Python 2.7, you can use a dictionary with configuration details. PEP 391 contains a list of the mandatory and optional elements in the configuration dictionary.

```
import logging
from logging.config import dictConfig

logging_config = dict(
    version = 1,
    formatters = {
        'f': {'format':
              '%(asctime)s %(name)-12s %(levelname)-8s %(message)s'}
```

```
                %(asctime)s %(name)-12s %(levelname)-8s %(message)s'}
        },
    handlers = {
        'h': {'class': 'logging.StreamHandler',
               'formatter': 'f',
               'level': logging.DEBUG}
        },
    root = {
        'handlers': ['h'],
        'level': logging.DEBUG,
        },
)

dictConfig(logging_config)

logger = logging.getLogger()
logger.debug('often makes a very good meal of %s', 'visiting tourists')
```

Logging exceptions

If you want to log exceptions you can and should make use of the logging.exception(msg) method:

```
>>> import logging
>>> logging.basicConfig()
>>> try:
...     raise Exception('foo')
... except:
...     logging.exception('bar')
...
ERROR:root:bar
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
Exception: foo
```

Do not pass the exception as argument:

As logging.exception(msg) expects a msg arg, it is a common pitfall to pass the exception into the logging call like this:

```
>>> try:
...     raise Exception('foo')
... except Exception as e:
...     logging.exception(e)
...
ERROR:root:foo
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
Exception: foo
```

While it might look as if this is the right thing to do at first, it is actually problematic due to the reason how exceptions and various encoding work together in the logging module:

```
>>> try:
...     raise Exception(u'föö')
... except Exception as e:
...     logging.exception(e)
...
Traceback (most recent call last):
  File "/.../python2.7/logging/__init__.py", line 861, in emit
    msg = self.format(record)
  File "/.../python2.7/logging/__init__.py", line 734, in format
    return fmt.format(record)
  File "/.../python2.7/logging/__init__.py", line 469, in format
    s = self._fmt % record.__dict__
UnicodeEncodeError: 'ascii' codec can't encode characters in position 1-2: ordinal not in range(
Logged from file <stdin>, line 4
```

Trying to log an exception that contains unicode chars, this way will fail miserably . It will hide the stacktrace of the original exception by overriding it with a new one that is raised during formatting of your logging.exception(e) call.

Obviously, in your own code, you might be aware of the encoding in exceptions. However, 3rd party libs might handle this in a different way.

Correct Usage:

If instead of the exception you just pass a message and let python do its magic, it will work:

```
>>> try:
...     raise Exception(u'föö')
... except Exception as e:
...     logging.exception('bar')
...
```

```
ERROR:root:bar
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
Exception: f\xf6\xf6
```

As you can see we don't actually use `e` in that case, the call to `logging.exception(...)` magically formats the most recent exception.

Logging exceptions with non ERROR log levels

If you want to log an exception with another log level than ERROR, you can use the the `exc_info` argument of the default loggers:

```
logging.debug('exception occurred', exc_info=1)
logging.info('exception occurred', exc_info=1)
logging.warning('exception occurred', exc_info=1)
```

Accessing the exception's message

Be aware that libraries out there might throw exceptions with messages as any of unicode or (utf-8 if you're lucky) byte-strings. If you really need to access an exception's text, the only reliable way, that will always work, is to use `repr(e)` or the `%r` string formatting:

```
>>> try:
...     raise Exception(u'föö')
... except Exception as e:
...     logging.exception('received this exception: %r' % e)
...
ERROR:root:received this exception: Exception(u'f\xf6\xf6',)
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
Exception: f\xf6\xf6
```

Syntax

Parameters

Remarks