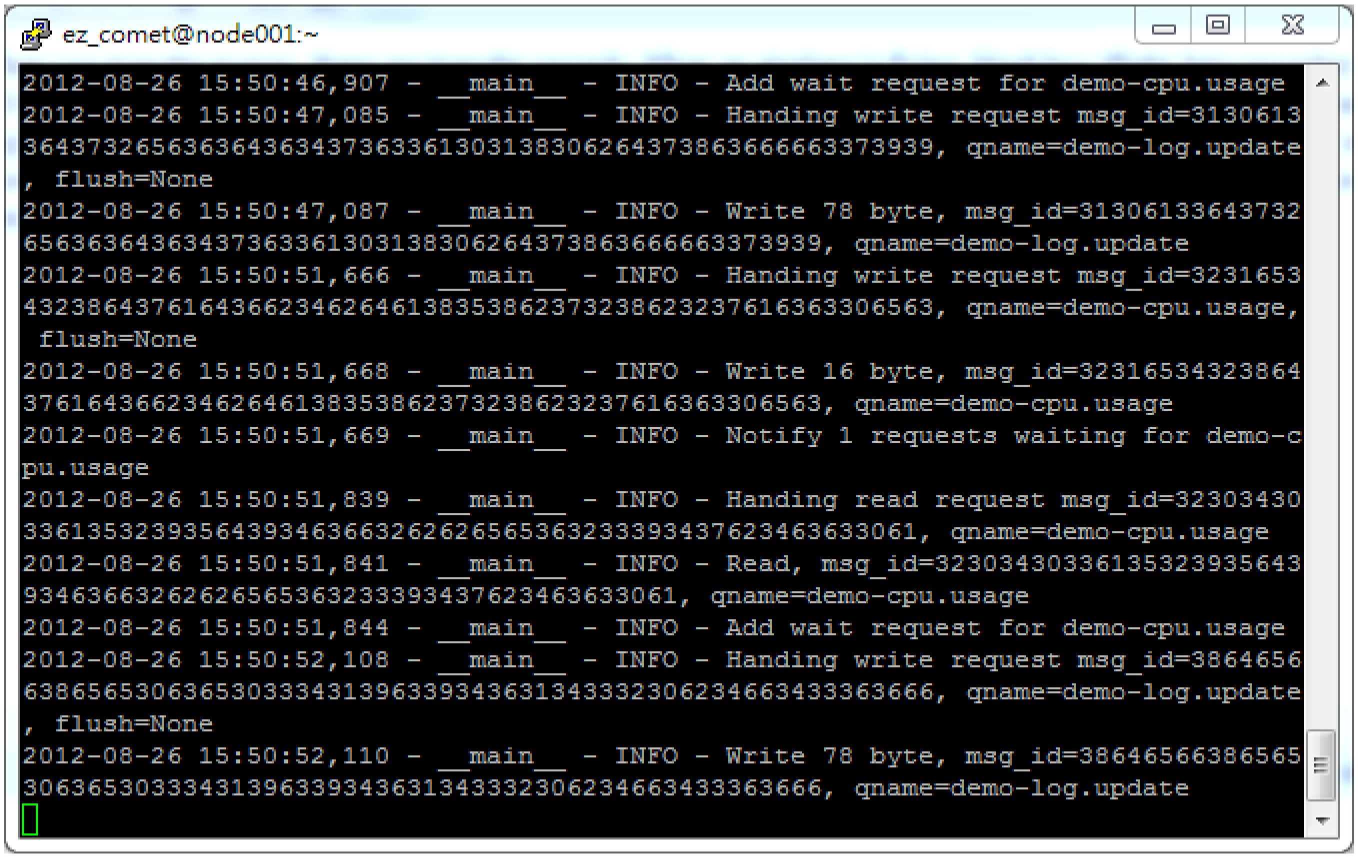
# Good logging practice in Python

Aug 26, 2012
12 minute read

In reality, logging is important. When you transfer money, there are transfer records. When an airplane is flying, black box (flight data recorder) is recording everything. If something goes wrong, people can read the log and has a chance to figure out what happened. Likewise, logging is important for system developing, debugging and running. When a program crashes, if there is no logging record, you have little chance to understand what happened. For example, when you are writing a server, logging is necessary. Following screenshot is the log file of a EZComet.com server.



```
2012-08-26 15:50:46,907 - __main__ - INFO - Add wait request for demo-cpu.usage
2012-08-26 15:50:47,085 - __main__ - INFO - Handing write request msg_id=3130613
36437326563636436343736336130313830626437386366666633373939, qname=demo-log.update
, flush=None
2012-08-26 15:50:47,087 - __main__ - INFO - Write 78 byte, msg_id=31306133643732
656363643634373633613031383062643738636666663373939, qname=demo-log.update
2012-08-26 15:50:51,666 - __main__ - INFO - Handing write request msg_id=3231653
43238643761643662346264613835386237323862323761636330656 3, qname=demo-cpu.usage,
 flush=None
2012-08-26 15:50:51,668 - __main__ - INFO - Write 16 byte, msg_id=32316534323864
37616436623462646138353862373238623237616363306563, qname=demo-cpu.usage
2012-08-26 15:50:51,669 - __main__ - INFO - Notify 1 requests waiting for demo-c
pu.usage
2012-08-26 15:50:51,839 - __main__ - INFO - Handing read request msg_id=32303430
33613532393564393463663262626565363233393437623463633061, qname=demo-cpu.usage
2012-08-26 15:50:51,841 - __main__ - INFO - Read, msg_id=32303430336135323935643
9346366326262656536323339343762346363306 1, qname=demo-cpu.usage
2012-08-26 15:50:51,844 - __main__ - INFO - Add wait request for demo-cpu.usage
2012-08-26 15:50:52,108 - __main__ - INFO - Handing write request msg_id=3864656
63865653063653033343139633934363134333230623466343336366 6, qname=demo-log.update
, flush=None
2012-08-26 15:50:52,110 - __main__ - INFO - Write 78 byte, msg_id=38646566386565
30636530333431396339343631343332306234663433363666, qname=demo-log.update
```

Without the log, I can hardly know what’s wrong if a service goes down. Not only for the servers, logging is also important for desktop GUI applications. For instance, when your program crashes on your customer’s PC, you can ask them to send the log files to you, and you may can figure why. Trust me, you will never know what kind of strange issues there will be in different PC environments. I once received an error log report like this

```
2011-08-22 17:52:54,828 - root - ERROR - [Errno 10104] getaddrinfo failed
Traceback (most recent call last):
  File "<string>", line 124, in main
  File "<string>", line 20, in __init__
  File "h:\workspace\project\build\pyi.win32\mrdj\outPYZ1.pyz/wx._core", line
7978, in __init__
  File "h:\workspace\project\build\pyi.win32\mrdj\outPYZ1.pyz/wx._core", line
7552, in _BootstrapApp
  File "<string>", line 84, in OnInit
  File
"h:\workspace\project\build\pyi.win32\mrdj\outPYZ1.pyz/twisted.internet.wxrea
ctor", line 175, in install
  File
"h:\workspace\project\build\pyi.win32\mrdj\outPYZ1.pyz/twisted.internet._thre
adedselect", line 106, in __init__
  File
"h:\workspace\project\build\pyi.win32\mrdj\outPYZ1.pyz/twisted.internet.base"
, line 488, in __init__
  File
"h:\workspace\project\build\pyi.win32\mrdj\outPYZ1.pyz/twisted.internet.posix
base", line 266, in installWaker
  File
"h:\workspace\project\build\pyi.win32\mrdj\outPYZ1.pyz/twisted.internet.posix
base", line 74, in __init__
  File "h:\workspace\project\build\pyi.win32\mrdj\outPYZ1.pyz/socket", line
224, in meth
gaierror: [Errno 10104] getaddrinfo failed
```

And eventually figure out that the customer PC is infected by a virus which makes call to gethostname failed. See, how can you even know this if there is no log to read?

## print is not a good idea

Although logging is important, not all developers know how to use them correctly. I saw some developers insert print statements when developing and remove those statements when it is finished. It may looks like this

```
print 'Start reading database'
records = model.read_recrods()
print '# records', records
print 'Updating record ...'
model.update_records(records)
print 'done'
```

It works when the program is a simple script, but for complex systems, you better not to use this approach. First of all, you cannot leave only important messages in the log, you may see a lots of garbage messages in the log, but can’t find anything useful.You also cannot control those print statements without modifying code, you may forgot to remove those unused prints. And all printed messages go into stdout, which is bad when you have data to output to stdout. Of course you can print messages to stderr, but still, it is not a good practice to use print for logging.

## Use Python standard logging module

So, how do you do logging correctly? It's easy, use the standard Python logging module. Thanks to Python community, logging is a standard module, it was well designed to be easy-to-use and very flexible. You can use the logging system like this

```
import logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

logger.info('Start reading database')
# read database here
records = {'john': 55, 'tom': 66}
logger.debug('Records: %s', records)
logger.info('Updating records ...')
# update records here
logger.info('Finish updating records')
```

You can run it and see

```
INFO:__main__:Start reading database
INFO:__main__:Updating records ...
INFO:__main__:Finish updating records
```

What's different between the "print" approach you asked. Well, of course there are benefits:

- You can control message level and filter out not important ones
- You can decide where and how to output later

There are different importance levels you can use, debug, info, warning, error and critical. By giving different level to logger or handler, you can write only error messages to specific log file, or record debug details when debugging. Let's change the logger level to DEBUG and see the output again

```
logging.basicConfig(level=logging.DEBUG)
```

The output:

```
INFO:__main__:Start reading database
DEBUG:__main__:Records: {'john': 55, 'tom': 66}
INFO:__main__:Updating records ...
INFO:__main__:Finish updating records
```

As you can see, we adjust the logger level to DEBUG, then debug records appear in output. You can also decide how these messages are processed. For example, you can use a FileHandler to write records to a file.

```
import logging

logger = logging.getLogger(__name__)
logger.setLevel(logging.INFO)

# create a file handler
```

```
handler = logging.FileHandler('hello.log')
handler.setLevel(logging.INFO)

# create a logging format
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s -
%(message)s')
handler.setFormatter(formatter)

# add the handlers to the logger
logger.addHandler(handler)

logger.info('Hello baby')
```

There are different handlers, you can also send records to you mailbox or even a to a remote server. You can also write your own custom logging handler. I'm not going to tell you details, please reference to official documents: Basic Tutorial, Advanced Tutorial and Logging Cookbook.

## Write logging records everywhere with proper level

With flexibility of the logging module, you can write logging record everywhere with proper level and configure them later. What is the proper level to use, you may ask. Here I share my experience.

In most cases, you don't want to read too much details in the log file. Therefore, debug level is only enabled when you are debugging. I use debug level only for detail debugging information, especially when the data is big or the frequency is high, such as records of algorithm internal state changes in a for-loop.

```
def complex_algorithm(items):
    for i, item in enumerate(items):
        # do some complex algorithm computation
        logger.debug('%s iteration, item=%s', i, item)
```

I use info level for routines, for example, handling requests or server state changed.

```
def handle_request(request):
    logger.info('Handling request %s', request)
    # handle request here
    result = 'result'
    logger.info('Return result: %s', result)

def start_service():
    logger.info('Starting service at port %s ...', port)
    service.start()
    logger.info('Service is started')
```

I use warning when it is important, but not an error, for example, when a user attempts to login with wrong password or connection is slow.

```
def authenticate(user_name, password, ip_address):
    if user_name != USER_NAME and password != PASSWORD:
        logger.warn('Login attempt to %s from IP %s', user_name, ip_address)
        return False
    # do authentication here
```

I use error level when something is wrong, for example, an exception is thrown, IO operation failure or connectivity issue.

```
def get_user_by_id(user_id):
    user = db.read_user(user_id)
    if user is None:
        logger.error('Cannot find user with user_id=%s', user_id)
        return user
    return user
```

I seldom use critical, you can use it when something really bad happen, for example, out of memory, disk is full or a nuclear meltdown (Hope that will not happen :S).

## Use __name__ as the logger name

You don't have to set the logger name as __name__, but by doing that, it brings us some benefits. The variable __name__ is current module name in Python. For example, you call logger.getLogger(__name__) in a module "foo.bar.my_module", then it is logger.getLogger("foo.bar.my_module"). When you need to configure the logger, you can configure to "foo", then all modules in "foo" packages shares same configuration. You can also understand what is the module of message when reading the log.

## Capture exceptions and record them with traceback

It is always a good practice to record when something goes wrong, but it won't be helpful if there is no traceback. You should capture exceptions and record them with traceback. Following is an example:

```
try:
    open('/path/to/does/not/exist', 'rb')
except (SystemExit, KeyboardInterrupt):
    raise
except Exception, e:
    logger.error('Failed to open file', exc_info=True)
```

By calling logger methods with exc_info=True parameter, traceback is dumped to the logger. As you can see the result

```
ERROR:__main__:Failed to open file
Traceback (most recent call last):
  File "example.py", line 6, in <module>
    open('/path/to/does/not/exist', 'rb')
IOError: [Errno 2] No such file or directory: '/path/to/does/not/exist'
```

You can also call logger.exception(msg, *args), it equals to logger.error(msg, exc_info=True, *args).

## Do not get logger at the module level unless disable_existing_loggers is False

You can see a lots of example out there (including this article, I did it just for giving example in short) get logger at module level. They looks harmless, but actually, there is a pitfall – Python logging module respects all created logger before you load the configuration from a file, if you get logger at the module level like this

my_module.py

```
import logging

logger = logging.getLogger(__name__)

def foo():
    logger.info('Hi, foo')

class Bar(object):
    def bar(self):
        logger.info('Hi, bar')
```

main.py

```
import logging

# load my module
import my_module

# load the logging configuration
logging.config.fileConfig('logging.ini')

my_module.foo()
bar = my_module.Bar()
bar.bar()
```

logging.ini

```
[loggers]
keys=root

[handlers]
keys=consoleHandler

[formatters]
keys=simpleFormatter

[logger_root]
level=DEBUG
```

```
handlers=consoleHandler

[handler_consoleHandler]
class=StreamHandler
level=DEBUG
formatter=simpleFormatter
args=(sys.stdout,)

[formatter_simpleFormatter]
format=%(asctime)s - %(name)s - %(levelname)s - %(message)s
datefmt=
```

And you expect to see the records appear in log, but you will see nothing. Why? Because you create the logger at module level, you then import the module before you load the logging configuration from a file. The logging.fileConfig and logging.dictConfig disables existing loggers by default. So, those setting in file will not be applied to your logger. It’s better to get the logger when you need it. It’s cheap to create or get a logger. You can write the code like this:

```
import logging

def foo():
    logger = logging.getLogger(__name__)
    logger.info('Hi, foo')

class Bar(object):
    def __init__(self, logger=None):
        self.logger = logger or logging.getLogger(__name__)

    def bar(self):
        self.logger.info('Hi, bar')
```

By doing that, the loggers will be created after you load the configuration. The setting will be applied correctly.

Since Python2.7, a new argument name “disable_existing_loggers” to fileConfig and dictConfig (as a parameter in schema) is added, by setting it to False, problem mentioned above can be solved. For example:

```
import logging
import logging.config

logger = logging.getLogger(__name__)

# load config from file
# logging.config.fileConfig('logging.ini', disable_existing_loggers=False)
# or, for dictConfig
logging.config.dictConfig({
    'version': 1,
    'disable_existing_loggers': False,  # this fixes the problem
    'formatters': {
        'standard': {
            'format': '%(asctime)s [%(levelname)s] %(name)s: %(message)s'
        },
    },
    'handlers': {
        'default': {
            'level':'INFO',
            'class':'logging.StreamHandler',
        },
    },
    'loggers': {
        '': {
            'handlers': ['default'],
            'level': 'INFO',
            'propagate': True
        }
    }
})

logger.info('It works!')
```

## Use JSON or YAML logging configuration

You can configure your logging system in Python code, but it is not flexible. It's better to use a logging configuration file. After Python 2.7, you can load logging configuration from a dict. It means you can load the logging configuration from a JSON or YAML file. Although you can use the old .ini style logging configuration, it is difficult to read and write. Here I show you an logging configuration example in JSON or YAML

logging.json

```
{
    "version": 1,
    "disable_existing_loggers": false,
    "formatters": {
        "simple": {
            "format": "%(asctime)s - %(name)s - %(levelname)s - %(message)s"
        }
    },

    "handlers": {
        "console": {
            "class": "logging.StreamHandler",
            "level": "DEBUG",
            "formatter": "simple",
            "stream": "ext://sys.stdout"
        },

        "info_file_handler": {
            "class": "logging.handlers.RotatingFileHandler",
            "level": "INFO",
            "formatter": "simple",
            "filename": "info.log",
            "maxBytes": 10485760,
            "backupCount": 20,
            "encoding": "utf8"
        },
        "error_file_handler": {
            "class": "logging.handlers.RotatingFileHandler",
            "level": "ERROR",
            "formatter": "simple",
            "filename": "errors.log",
            "maxBytes": 10485760,
            "backupCount": 20,
            "encoding": "utf8"
        }
    },

    "loggers": {
        "my_module": {
            "level": "ERROR",
            "handlers": ["console"],
            "propagate": "no"
        }
    },

    "root": {
        "level": "INFO",
        "handlers": ["console", "info_file_handler", "error_file_handler"]
    }
}
```

logging.yaml

```
---
version: 1
disable_existing_loggers: False
formatters:
    simple:
        format: "%(asctime)s - %(name)s - %(levelname)s - %(message)s"

handlers:
    console:
        class: logging.StreamHandler
        level: DEBUG
        formatter: simple
        stream: ext://sys.stdout

    info_file_handler:
        class: logging.handlers.RotatingFileHandler
        level: INFO
        formatter: simple
        filename: info.log
        maxBytes: 10485760 # 10MB
        backupCount: 20
        encoding: utf8

    error_file_handler:
        class: logging.handlers.RotatingFileHandler
        level: ERROR
        formatter: simple
        filename: errors.log
        maxBytes: 10485760 # 10MB
        backupCount: 20
        encoding: utf8

loggers:
    my_module:
        level: ERROR
        handlers: [console]
        propagate: no

root:
    level: INFO
    handlers: [console, info_file_handler, error_file_handler]
...
```

Following recipe shows you how to read logging configuration from a JSON file:

```
import os
import json
import logging.config

def setup_logging(
    default_path='logging.json',
    default_level=logging.INFO,
    env_key='LOG_CFG'
):
    """Setup logging configuration

    """
    path = default_path
    value = os.getenv(env_key, None)
    if value:
        path = value
    if os.path.exists(path):
        with open(path, 'rt') as f:
            config = json.load(f)
        logging.config.dictConfig(config)
    else:
        logging.basicConfig(level=default_level)
```

One advantage of using JSON configuration is that the json is a standard library, you don't need to install it. But personally, I prefer YAML. It's very clear to read and easy to write. You can also load the YAML configuration with following recipes

```
import os
import logging.config

import yaml

def setup_logging(
    default_path='logging.yaml',
    default_level=logging.INFO,
    env_key='LOG_CFG'
):
    """Setup logging configuration
    """
    path = default_path
    value = os.getenv(env_key, None)
    if value:
        path = value
    if os.path.exists(path):
        with open(path, 'rt') as f:
            config = yaml.safe_load(f.read())
        logging.config.dictConfig(config)
    else:
        logging.basicConfig(level=default_level)
```

Now, to setup logging, you can call setup_logging when starting your program. It reads logging.json or logging.yaml by default. You can also set LOG_CFG environment variable to load the logging configuration from specific path. For example,

```
LOG_CFG=my_logging.json python my_server.py
```

or if you prefer YAML

```
LOG_CFG=my_logging.yaml python my_server.py
```

## Use rotating file handler

If you use FileHandler for writing logs, the size of log file will grow with time. Someday, it will occupy all of your disk. In order to avoid that situation, you should use RotatingFileHandler instead of FileHandler in production environment.

## Setup a central log server when you have multiple servers

When you have multiple servers and different log files. You can setup a central log system to collect all important (warning and error messages in most cases). Then you can monitor it easily and notice what's wrong in your system.

## Conclusions

I'm glad that Python logging library is nicely designed, and the best part is that it is a standard library, you don't have to choose. It is flexible, you can write your own handlers and filters. There are also third-party handlers such as ZeroMQ logging handler provided by pyzmq, it allows you to send logging messages through a zmq socket. If you don't know how to use the logging system correctly, this article might be helpful. With good logging practice, you can find issues in your system easier. It's a nice investment, don't you buy it? :D