
로깅 요리책

릴리스 3.14.0a1

Guido van Rossum and the Python development team

10월 27, 2024

Contents

1	여러 모듈에서 로깅 사용하기	2
2	여러 스레드에서 로깅 하기	4
3	다중 처리기 및 포매터	5
4	여러 대상으로 로깅 하기	6
5	Custom handling of levels	7
6	구성 서버 예제	10
7	블록 하는 처리기 다루기	11
8	네트워크에서 로깅 이벤트 보내고 받기	12
8.1	Running a logging socket listener in production	14
9	로그 출력에 문맥 정보 추가	15
9.1	문맥 정보 전달에 LoggerAdapters 사용하기	15
9.2	문맥 정보 전달에 필터 사용하기	16
10	Use of contextvars	18
11	Imparting contextual information in handlers	22
12	여러 프로세스에서 단일 파일에 로깅 하기	22
12.1	concurrent.futures.ProcessPoolExecutor 사용하기	26
12.2	Deploying Web applications using Gunicorn and uWSGI	27
13	파일 회전 사용하기	27
14	대체 포매팅 스타일 사용하기	28
15	사용자 정의 LogRecord	30
16	Subclassing QueueHandler and QueueListener- a ZeroMQ example	31
16.1	Subclass QueueHandler	31
16.2	Subclass QueueListener	32
17	Subclassing QueueHandler and QueueListener- a pynng example	32
17.1	Subclass QueueListener	32
17.2	Subclass QueueHandler	33

18	딕셔너리 기반 구성의 예	35
19	rotator와 namer를 사용해서 로그 회전 처리하기	36
20	좀 더 정교한 multiprocessing 예제	37
21	SysLogHandler로 전송된 메시지에 BOM 삽입하기	41
22	구조적 로깅 구현	41
23	dictConfig()로 처리기를 사용자 정의하기	43
24	응용 프로그램 전체에서 특정 포맷 스타일 사용하기	45
	24.1 LogRecord 팩토리 사용	45
	24.2 사용자 정의 메시지 객체 사용	45
25	dictConfig()로 필터 구성하기	46
26	사용자 정의된 예외 포매팅	47
27	로깅 메시지 말하기	48
28	로깅 메시지를 버퍼링하고 조건부 출력하기	49
29	Sending logging messages to email, with buffering	51
30	구성을 통해 UTC(GMT)로 시간을 포맷하기	53
31	선택적 로깅을 위해 컨텍스트 관리자 사용하기	54
32	CLI 응용 프로그램 시작 템플릿	55
33	로깅을 위한 Qt GUI	58
34	Logging to syslog with RFC5424 support	62
35	How to treat a logger like an output stream	64
36	Patterns to avoid	66
	36.1 Opening the same log file multiple times	66
	36.2 Using loggers as attributes in a class or passing them as parameters	67
	36.3 Adding handlers other than NullHandler to a logger in a library	67
	36.4 Creating a lot of loggers	67
37	Other resources	67
	색인	68

저자
 Vinay Sajip <vinay_sajip at red-dove dot com>

This page contains a number of recipes related to logging, which have been found useful in the past. For links to tutorial and reference information, please see *Other resources*.

1 여러 모듈에서 로깅 사용하기

logging.getLogger('someLogger')를 여러 번 호출하면 같은 로거 객체에 대한 참조가 반환됩니다. 같은 모듈 내에서뿐만 아니라, 같은 파이썬 인터프리터 프로세스에 있는 한, 여러 모듈에서도 마찬가지입니다. 참조가 같은 객체를 가리킨다는 것에 더해, 응용 프로그램 코드는 하나의 모듈에서 부모 로거를

정의 및 구성하고 별도의 모듈에서 자식 로거를 생성(구성하지 않음) 할 수 있으며, 자식에 대한 모든 로거 호출은 부모로 전달됩니다. 다음은 메인 모듈입니다:

```
import logging
import auxiliary_module

# create logger with 'spam_application'
logger = logging.getLogger('spam_application')
logger.setLevel(logging.DEBUG)
# create file handler which logs even debug messages
fh = logging.FileHandler('spam.log')
fh.setLevel(logging.DEBUG)
# create console handler with a higher log level
ch = logging.StreamHandler()
ch.setLevel(logging.ERROR)
# create formatter and add it to the handlers
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s
↪')
fh.setFormatter(formatter)
ch.setFormatter(formatter)
# add the handlers to the logger
logger.addHandler(fh)
logger.addHandler(ch)

logger.info('creating an instance of auxiliary_module.Auxiliary')
a = auxiliary_module.Auxiliary()
logger.info('created an instance of auxiliary_module.Auxiliary')
logger.info('calling auxiliary_module.Auxiliary.do_something()')
a.do_something()
logger.info('finished auxiliary_module.Auxiliary.do_something()')
logger.info('calling auxiliary_module.some_function()')
auxiliary_module.some_function()
logger.info('done with auxiliary_module.some_function()')
```

다음은 보조 모듈입니다:

```
import logging

# create logger
module_logger = logging.getLogger('spam_application.auxiliary')

class Auxiliary:
    def __init__(self):
        self.logger = logging.getLogger('spam_application.auxiliary.Auxiliary')
        self.logger.info('creating an instance of Auxiliary')

    def do_something(self):
        self.logger.info('doing something')
        a = 1 + 1
        self.logger.info('done doing something')

def some_function():
    module_logger.info('received a call to "some_function"')
```

출력은 이렇게 됩니다:

```
2005-03-23 23:47:11,663 - spam_application - INFO -
  creating an instance of auxiliary_module.Auxiliary
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
2005-03-23 23:47:11,665 - spam_application.auxiliary.Auxiliary - INFO -
    creating an instance of Auxiliary
2005-03-23 23:47:11,665 - spam_application - INFO -
    created an instance of auxiliary_module.Auxiliary
2005-03-23 23:47:11,668 - spam_application - INFO -
    calling auxiliary_module.Auxiliary.do_something
2005-03-23 23:47:11,668 - spam_application.auxiliary.Auxiliary - INFO -
    doing something
2005-03-23 23:47:11,669 - spam_application.auxiliary.Auxiliary - INFO -
    done doing something
2005-03-23 23:47:11,670 - spam_application - INFO -
    finished auxiliary_module.Auxiliary.do_something
2005-03-23 23:47:11,671 - spam_application - INFO -
    calling auxiliary_module.some_function()
2005-03-23 23:47:11,672 - spam_application.auxiliary - INFO -
    received a call to 'some_function'
2005-03-23 23:47:11,673 - spam_application - INFO -
    done with auxiliary_module.some_function()
```

2 여러 스레드에서 로깅 하기

여러 스레드에서 로깅 하는데 특별한 노력이 필요하지는 않습니다. 다음 예제에서는 메인 (최초) 스레드와 다른 스레드에서의 로깅을 보여줍니다:

```
import logging
import threading
import time

def worker(arg):
    while not arg['stop']:
        logging.debug('Hi from myfunc')
        time.sleep(0.5)

def main():
    logging.basicConfig(level=logging.DEBUG, format='%(relativeCreated)6d
↪%(threadName)s %(message)s')
    info = {'stop': False}
    thread = threading.Thread(target=worker, args=(info,))
    thread.start()
    while True:
        try:
            logging.debug('Hello from main')
            time.sleep(0.75)
        except KeyboardInterrupt:
            info['stop'] = True
            break
    thread.join()

if __name__ == '__main__':
    main()
```

실행하면 스크립트는 다음과 같이 인쇄합니다:

```
0 Thread-1 Hi from myfunc
3 MainThread Hello from main
```

(다음 페이지에 계속)

```

505 Thread-1 Hi from myfunc
755 MainThread Hello from main
1007 Thread-1 Hi from myfunc
1507 MainThread Hello from main
1508 Thread-1 Hi from myfunc
2010 Thread-1 Hi from myfunc
2258 MainThread Hello from main
2512 Thread-1 Hi from myfunc
3009 MainThread Hello from main
3013 Thread-1 Hi from myfunc
3515 Thread-1 Hi from myfunc
3761 MainThread Hello from main
4017 Thread-1 Hi from myfunc
4513 MainThread Hello from main
4518 Thread-1 Hi from myfunc

```

예상대로 로그 출력이 산재해 있음을 볼 수 있습니다. 물론, 이 방법은 여기에 표시된 것보다 많은 스레드에서도 작동합니다.

3 다중 처리기 및 포맷터

로거는 일반 파이썬 객체입니다. `addHandler()` 메서드에는 추가할 수 있는 처리기의 수에 대한 최소 또는 최대 할당량이 없습니다. 때로는 응용 프로그램이 모든 심각도의 모든 메시지를 텍스트 파일에 기록하는 동시에, 예러 또는 그 이상을 콘솔에 기록하는 것이 유용 할 수 있습니다. 이렇게 설정하려면, 적절한 처리기를 구성하기만 하면 됩니다. 응용 프로그램 코드의 로깅 호출은 변경되지 않습니다. 다음은 앞의 간단한 모듈 기반 구성 예제를 약간 수정 한 것입니다:

```

import logging

logger = logging.getLogger('simple_example')
logger.setLevel(logging.DEBUG)
# create file handler which logs even debug messages
fh = logging.FileHandler('spam.log')
fh.setLevel(logging.DEBUG)
# create console handler with a higher log level
ch = logging.StreamHandler()
ch.setLevel(logging.ERROR)
# create formatter and add it to the handlers
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s
↪')
ch.setFormatter(formatter)
fh.setFormatter(formatter)
# add the handlers to logger
logger.addHandler(ch)
logger.addHandler(fh)

# 'application' code
logger.debug('debug message')
logger.info('info message')
logger.warning('warn message')
logger.error('error message')
logger.critical('critical message')

```

‘응용 프로그램’ 코드는 여러 처리기에 신경 쓰지 않습니다. 변경된 것은 `fh` 라는 새로운 처리기의 추가 및 구성뿐입니다.

중요도가 높거나 낮은 필터를 사용하여 새 처리기를 만드는 기능은 응용 프로그램을 작성하고 테스트할

때 매우 유용합니다. 디버깅을 위해 많은 `print` 문을 사용하는 대신에 `logger.debug` 를 사용하십시오: 나중에 삭제하거나 주석 처리해야 할 `print` 문과 달리, `logger.debug` 문은 소스 코드에서 그대로 유지될 수 있고, 그들을 다시 필요로 할 때까지 휴면 상태로 남아 있습니다. 그때, 필요한 유일한 변경은 로거 또는 처리기의 심각도 수준을 `DEBUG`로 수정하는 것입니다.

4 여러 대상으로 로깅 하기

다른 메시지 포맷으로 다른 상황에서 콘솔과 파일에 기록하려고 한다고 가정 해 봅시다. `DEBUG` 이상 수준의 메시지를 파일에 기록하고, 수준 `INFO` 이상인 메시지를 콘솔에 기록하려고 한다고 가정 해보십시오. 또한, 타임스탬프가 파일에는 포함되어야 하지만, 콘솔 메시지에는 없어야 한다고 가정합니다. 이렇게 하면 됩니다:

```
import logging

# set up logging to file - see previous section for more details
logging.basicConfig(level=logging.DEBUG,
                    format='%(asctime)s %(name)-12s %(levelname)-8s %(message)s',
                    datefmt='%m-%d %H:%M',
                    filename='/tmp/myapp.log',
                    filemode='w')

# define a Handler which writes INFO messages or higher to the sys.stderr
console = logging.StreamHandler()
console.setLevel(logging.INFO)
# set a format which is simpler for console use
formatter = logging.Formatter('%(name)-12s: %(levelname)-8s %(message)s')
# tell the handler to use this format
console.setFormatter(formatter)
# add the handler to the root logger
logging.getLogger('').addHandler(console)

# Now, we can log to the root logger, or any other logger. First the root...
logging.info('Jackdaws love my big sphinx of quartz.')

# Now, define a couple of other loggers which might represent areas in your
# application:

logger1 = logging.getLogger('myapp.area1')
logger2 = logging.getLogger('myapp.area2')

logger1.debug('Quick zephyrs blow, vexing daft Jim.')
logger1.info('How quickly daft jumping zebras vex.')
logger2.warning('Jail zesty vixen who grabbed pay from quack.')
logger2.error('The five boxing wizards jump quickly.')
```

실행하면 콘솔에는 다음과 같이 출력됩니다.

```
root          : INFO      Jackdaws love my big sphinx of quartz.
myapp.area1   : INFO      How quickly daft jumping zebras vex.
myapp.area2   : WARNING   Jail zesty vixen who grabbed pay from quack.
myapp.area2   : ERROR     The five boxing wizards jump quickly.
```

파일에는 이렇게 기록됩니다.

```
10-22 22:19 root          INFO      Jackdaws love my big sphinx of quartz.
10-22 22:19 myapp.area1  DEBUG     Quick zephyrs blow, vexing daft Jim.
10-22 22:19 myapp.area1  INFO      How quickly daft jumping zebras vex.
10-22 22:19 myapp.area2  WARNING   Jail zesty vixen who grabbed pay from quack.
```

(다음 페이지에 계속)

```
10-22 22:19 myapp.area2 ERROR The five boxing wizards jump quickly.
```

보시다시피 DEBUG 메시지는 파일에만 표시됩니다. 다른 메시지는 두 목적으로 전송됩니다.

이 예제는 콘솔과 파일 처리기를 사용하지만, 여러분이 선택하는 처리기의 수나 조합에 제약이 없습니다.

Note that the above choice of log filename `/tmp/myapp.log` implies use of a standard location for temporary files on POSIX systems. On Windows, you may need to choose a different directory name for the log - just ensure that the directory exists and that you have the permissions to create and update files in it.

5 Custom handling of levels

Sometimes, you might want to do something slightly different from the standard handling of levels in handlers, where all levels above a threshold get processed by a handler. To do this, you need to use filters. Let's look at a scenario where you want to arrange things as follows:

- Send messages of severity INFO and WARNING to `sys.stdout`
- Send messages of severity ERROR and above to `sys.stderr`
- Send messages of severity DEBUG and above to file `app.log`

Suppose you configure logging with the following JSON:

```
{
  "version": 1,
  "disable_existing_loggers": false,
  "formatters": {
    "simple": {
      "format": "%(levelname)-8s - %(message)s"
    }
  },
  "handlers": {
    "stdout": {
      "class": "logging.StreamHandler",
      "level": "INFO",
      "formatter": "simple",
      "stream": "ext://sys.stdout"
    },
    "stderr": {
      "class": "logging.StreamHandler",
      "level": "ERROR",
      "formatter": "simple",
      "stream": "ext://sys.stderr"
    },
    "file": {
      "class": "logging.FileHandler",
      "formatter": "simple",
      "filename": "app.log",
      "mode": "w"
    }
  },
  "root": {
    "level": "DEBUG",
    "handlers": [
      "stderr",
      "stdout",
      "file"
    ]
  }
}
```

```

    ]
}
}

```

This configuration does *almost* what we want, except that `sys.stdout` would show messages of severity `ERROR` and only events of this severity and higher will be tracked as well as `INFO` and `WARNING` messages. To prevent this, we can set up a filter which excludes those messages and add it to the relevant handler. This can be configured by adding a `filters` section parallel to `formatters` and `handlers`:

```

{
  "filters": {
    "warnings_and_below": {
      "()" : "__main__.filter_maker",
      "level": "WARNING"
    }
  }
}

```

and changing the section on the `stdout` handler to add it:

```

{
  "stdout": {
    "class": "logging.StreamHandler",
    "level": "INFO",
    "formatter": "simple",
    "stream": "ext://sys.stdout",
    "filters": ["warnings_and_below"]
  }
}

```

A filter is just a function, so we can define the `filter_maker` (a factory function) as follows:

```

def filter_maker(level):
    level = getattr(logging, level)

    def filter(record):
        return record.levelno <= level

    return filter

```

This converts the string argument passed in to a numeric level, and returns a function which only returns `True` if the level of the passed in record is at or below the specified level. Note that in this example I have defined the `filter_maker` in a test script `main.py` that I run from the command line, so its module will be `__main__` - hence the `__main__.filter_maker` in the filter configuration. You will need to change that if you define it in a different module.

With the filter added, we can run `main.py`, which in full is:

```

import json
import logging
import logging.config

CONFIG = '''
{
  "version": 1,
  "disable_existing_loggers": false,
  "formatters": {

```

```

    "simple": {
        "format": "%(levelname)-8s - %(message)s"
    }
},
"filters": {
    "warnings_and_below": {
        "()" : "__main__.filter_maker",
        "level": "WARNING"
    }
},
"handlers": {
    "stdout": {
        "class": "logging.StreamHandler",
        "level": "INFO",
        "formatter": "simple",
        "stream": "ext://sys.stdout",
        "filters": ["warnings_and_below"]
    },
    "stderr": {
        "class": "logging.StreamHandler",
        "level": "ERROR",
        "formatter": "simple",
        "stream": "ext://sys.stderr"
    },
    "file": {
        "class": "logging.FileHandler",
        "formatter": "simple",
        "filename": "app.log",
        "mode": "w"
    }
},
"root": {
    "level": "DEBUG",
    "handlers": [
        "stderr",
        "stdout",
        "file"
    ]
}
}
'''

```

```

def filter_maker(level):
    level = getattr(logging, level)

    def filter(record):
        return record.levelno <= level

    return filter

```

```

logging.config.dictConfig(json.loads(CONFIG))
logging.debug('A DEBUG message')
logging.info('An INFO message')
logging.warning('A WARNING message')
logging.error('An ERROR message')
logging.critical('A CRITICAL message')

```

And after running it like this:

```
python main.py 2>stderr.log >stdout.log
```

We can see the results are as expected:

```
$ more *.log
:::
app.log
:::
DEBUG    - A DEBUG message
INFO     - An INFO message
WARNING  - A WARNING message
ERROR    - An ERROR message
CRITICAL - A CRITICAL message
:::
stderr.log
:::
ERROR    - An ERROR message
CRITICAL - A CRITICAL message
:::
stdout.log
:::
INFO     - An INFO message
WARNING  - A WARNING message
```

6 구성 서버 예제

다음은 로깅 구성 서버를 사용하는 모듈의 예입니다:

```
import logging
import logging.config
import time
import os

# read initial config file
logging.config.fileConfig('logging.conf')

# create and start listener on port 9999
t = logging.config.listen(9999)
t.start()

logger = logging.getLogger('simpleExample')

try:
    # loop through logging calls to see the difference
    # new configurations make, until Ctrl+C is pressed
    while True:
        logger.debug('debug message')
        logger.info('info message')
        logger.warning('warn message')
        logger.error('error message')
        logger.critical('critical message')
        time.sleep(5)
except KeyboardInterrupt:
    # cleanup
```

(다음 페이지에 계속)

```
logging.config.stopListening()
t.join()
```

다음은 파일 이름을 받아서, 그 파일을 새 로깅 구성으로 (이전 인코딩된 길이를 적절하게 앞에 붙여서) 서버로 보내는 스크립트입니다:

```
#!/usr/bin/env python
import socket, sys, struct

with open(sys.argv[1], 'rb') as f:
    data_to_send = f.read()

HOST = 'localhost'
PORT = 9999
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print('connecting...')
s.connect((HOST, PORT))
print('sending config...')
s.send(struct.pack('>L', len(data_to_send)))
s.send(data_to_send)
s.close()
print('complete')
```

7 블록 하는 처리기 다루기

Sometimes you have to get your logging handlers to do their work without blocking the thread you're logging from. This is common in web applications, though of course it also occurs in other scenarios.

흔히 느린 행동을 보이는 범인은 SMTPHandler 입니다: 개발자의 통제 밖에 있는 여러 가지 이유로, 전자 우편을 보내는 데 오랜 시간이 걸릴 수 있습니다 (예를 들어, 잘 동작하지 않는 메일 또는 네트워크 인프라). 그러나 거의 모든 네트워크 기반 처리기는 블록 할 수 있습니다. SocketHandler 작업도 너무 느린 DNS 질의를 이면에서 수행 할 수 있습니다 (그리고 이 질의는 여러분의 통제 밖에 있는, 파이썬 계층 아래의 소켓 라이브러리 코드 깊숙이 있을 수 있습니다).

한 가지 해결책은 두 부분으로 된 접근법을 사용하는 것입니다. 첫 번째 부분에서는, 성능이 중요한 스레드에서 액세스하는 로거에 QueueHandler 만 붙입니다. 그들은 단순히 큐에 씁니다. 충분한 용량으로 큐의 크기를 조정하거나, 크기의 상한이 없도록 초기화 할 수 있습니다. 큐에 대한 쓰기는 일반적으로 신속하게 받아들여지지만, 코드에서 예방책으로 queue.Full 예외를 잡아야 할 것입니다. 코드에 성능이 중요한 스레드가 있는 라이브러리 개발자인 경우, 코드를 사용할 다른 개발자의 이익을 위해 이것을 (여러분의 로거에 QueueHandlers 만 붙이라는 제안과 함께) 문서로 만들어야 합니다.

해결책의 두 번째 부분은 QueueListener 며, 이는 QueueHandler 에 상응하여 설계되었습니다. QueueListener 는 매우 간단합니다: 큐와 처리기를 넘겨주면 QueueHandlers (또는 LogRecords 의 다른 소스)에서 보낸 LogRecord 를 큐에서 수신하는 내부 스레드를 시작합니다. LogRecords 는 큐에서 제거되고 처리를 위해 처리기로 전달됩니다.

별도의 QueueListener 클래스를 사용하면 같은 인스턴스를 사용하여 여러 개의 QueueHandlers 를 처리할 수 있다는 장점이 있습니다. 이것은 특별한 이점 없이 처리기당 하나의 스레드를 먹게 되는 기존의 처리기 클래스의 스레드 버전을 만드는 것보다 자원 친화적입니다.

이 두 클래스를 사용하는 예제는 다음과 같습니다 (임포트 생략):

```
que = queue.Queue(-1) # no limit on size
queue_handler = QueueHandler(que)
handler = logging.StreamHandler()
listener = QueueListener(que, handler)
root = logging.getLogger()
root.addHandler(queue_handler)
```

(이전 페이지에서 계속)

```

formatter = logging.Formatter('%(threadName)s: %(message)s')
handler.setFormatter(formatter)
listener.start()
# The log output will display the thread which generated
# the event (the main thread) rather than the internal
# thread which monitors the internal queue. This is what
# you want to happen.
root.warning('Look out!')
listener.stop()

```

실행하면, 다음과 같은 결과를 만듭니다:

```
MainThread: Look out!
```

i 참고

Although the earlier discussion wasn't specifically talking about async code, but rather about slow logging handlers, it should be noted that when logging from async code, network and even file handlers could lead to problems (blocking the event loop) because some logging is done from `asyncio` internals. It might be best, if any async code is used in an application, to use the above approach for logging, so that any blocking code runs only in the `QueueListener` thread.

버전 3.5에서 변경: 파이썬 3.5 이전 버전에서는, `QueueListener` 는 항상 큐에서 받은 모든 메시지를 초기화될 때 제공된 모든 처리기로 전달했습니다. (이것은 큐가 채워질 때 수준 필터링이 모두 반대편에서 행해졌다고 가정했기 때문입니다.) 3.5 이후부터, 이 동작은 키워드 인자 `respect_handler_level=True` 를 리스너의 생성자에 전달함으로써 변경될 수 있습니다. 이렇게 할 때, 리스너는 각 메시지의 수준을 처리기의 수준과 비교하여, 적절한 메시지만 처리기에 전달되도록 합니다.

8 네트워크에서 로깅 이벤트 보내고 받기

네트워크를 통해 로깅 이벤트를 보내고, 받는 쪽에서 처리하려고 한다고 합시다. 이렇게 하는 간단한 방법은 `SocketHandler` 인스턴스를 보내는 쪽의 루트 로거에 연결하는 것입니다:

```

import logging, logging.handlers

rootLogger = logging.getLogger('')
rootLogger.setLevel(logging.DEBUG)
socketHandler = logging.handlers.SocketHandler('localhost',
                                               logging.handlers.DEFAULT_TCP_LOGGING_PORT)
# don't bother with a formatter, since a socket handler sends the event as
# an unformatted pickle
rootLogger.addHandler(socketHandler)

# Now, we can log to the root logger, or any other logger. First the root...
logging.info('Jackdaws love my big sphinx of quartz.')

# Now, define a couple of other loggers which might represent areas in your
# application:

logger1 = logging.getLogger('myapp.area1')
logger2 = logging.getLogger('myapp.area2')

logger1.debug('Quick zephyrs blow, vexing daft Jim.')
logger1.info('How quickly daft jumping zebras vex.')

```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
logger2.warning('Jail zesty vixen who grabbed pay from quack.')
logger2.error('The five boxing wizards jump quickly.')
```

수신 측에서는 socketserver 모듈을 사용하여 수신기를 구성할 수 있습니다. 기본적인 작업 예제는 다음과 같습니다:

```
import pickle
import logging
import logging.handlers
import socketserver
import struct

class LogRecordStreamHandler(socketserver.StreamRequestHandler):
    """Handler for a streaming logging request.

    This basically logs the record using whatever logging policy is
    configured locally.
    """

    def handle(self):
        """
        Handle multiple requests - each expected to be a 4-byte length,
        followed by the LogRecord in pickle format. Logs the record
        according to whatever policy is configured locally.
        """
        while True:
            chunk = self.connection.recv(4)
            if len(chunk) < 4:
                break
            slen = struct.unpack('>L', chunk)[0]
            chunk = self.connection.recv(slen)
            while len(chunk) < slen:
                chunk = chunk + self.connection.recv(slen - len(chunk))
            obj = self.unPickle(chunk)
            record = logging.makeLogRecord(obj)
            self.handleLogRecord(record)

    def unPickle(self, data):
        return pickle.loads(data)

    def handleLogRecord(self, record):
        # if a name is specified, we use the named logger rather than the one
        # implied by the record.
        if self.server.logname is not None:
            name = self.server.logname
        else:
            name = record.name
        logger = logging.getLogger(name)
        # N.B. EVERY record gets logged. This is because Logger.handle
        # is normally called AFTER logger-level filtering. If you want
        # to do filtering, do it at the client end to save wasting
        # cycles and network bandwidth!
        logger.handle(record)

class LogRecordSocketReceiver(socketserver.ThreadingTCPServer):
```

(다음 페이지에 계속)

```

"""
Simple TCP socket-based logging receiver suitable for testing.
"""

allow_reuse_address = True

def __init__(self, host='localhost',
             port=logging.handlers.DEFAULT_TCP_LOGGING_PORT,
             handler=LogRecordStreamHandler):
    socketserver.ThreadingTCPServer.__init__(self, (host, port), handler)
    self.abort = 0
    self.timeout = 1
    self.logname = None

def serve_until_stopped(self):
    import select
    abort = 0
    while not abort:
        rd, wr, ex = select.select([self.socket.fileno()],
                                  [], [],
                                  self.timeout)

        if rd:
            self.handle_request()
            abort = self.abort

def main():
    logging.basicConfig(
        format='%(relativeCreated)5d %(name)-15s %(levelname)-8s %(message)s')
    tcpserver = LogRecordSocketReceiver()
    print('About to start TCP server...')
    tcpserver.serve_until_stopped()

if __name__ == '__main__':
    main()

```

먼저 서버를 실행한 다음 클라이언트를 실행합니다. 클라이언트 쪽에서는 콘솔에 아무것도 인쇄되지 않습니다. 서버 측에서 다음과 같은 내용이 보여야 합니다.:

```

About to start TCP server...
59 root          INFO      Jackdaws love my big sphinx of quartz.
59 myapp.area1   DEBUG    Quick zephyrs blow, vexing daft Jim.
69 myapp.area1   INFO     How quickly daft jumping zebras vex.
69 myapp.area2   WARNING  Jail zesty vixen who grabbed pay from quack.
69 myapp.area2   ERROR    The five boxing wizards jump quickly.

```

Note that there are some security issues with pickle in some scenarios. If these affect you, you can use an alternative serialization scheme by overriding the `makePickle()` method and implementing your alternative there, as well as adapting the above script to use your alternative serialization.

8.1 Running a logging socket listener in production

To run a logging listener in production, you may need to use a process-management tool such as [Supervisor](#). [Here is a Gist](#) which provides the bare-bones files to run the above functionality using Supervisor. It consists of the following files:

File	Purpose
<code>prepare.sh</code>	A Bash script to prepare the environment for testing
<code>supervisor.conf</code>	The Supervisor configuration file, which has entries for the listener and a multi-process web application
<code>ensure_app.sh</code>	A Bash script to ensure that Supervisor is running with the above configuration
<code>log_listener.py</code>	The socket listener program which receives log events and records them to a file
<code>main.py</code>	A simple web application which performs logging via a socket connected to the listener
<code>webapp.json</code>	A JSON configuration file for the web application
<code>client.py</code>	A Python script to exercise the web application

The web application uses [Gunicorn](#), which is a popular web application server that starts multiple worker processes to handle requests. This example setup shows how the workers can write to the same log file without conflicting with one another — they all go through the socket listener.

To test these files, do the following in a POSIX environment:

1. Download the [Gist](#) as a ZIP archive using the *Download ZIP* button.
2. Unzip the above files from the archive into a scratch directory.
3. In the scratch directory, run `bash prepare.sh` to get things ready. This creates a `run` subdirectory to contain Supervisor-related and log files, and a `venv` subdirectory to contain a virtual environment into which `bottle`, `gunicorn` and `supervisor` are installed.
4. Run `bash ensure_app.sh` to ensure that Supervisor is running with the above configuration.
5. Run `venv/bin/python client.py` to exercise the web application, which will lead to records being written to the log.
6. Inspect the log files in the `run` subdirectory. You should see the most recent log lines in files matching the pattern `app.log*`. They won't be in any particular order, since they have been handled concurrently by different worker processes in a non-deterministic way.
7. You can shut down the listener and the web application by running `venv/bin/supervisorctl -c supervisor.conf shutdown`.

You may need to tweak the configuration files in the unlikely event that the configured ports clash with something else in your test environment.

9 로그 출력에 문맥 정보 추가

로깅 호출에 전달된 매개 변수 외에도 로깅 출력에 문맥 정보가 포함되기 원하는 경우가 있습니다. 예를 들어, 네트워크 응용 프로그램에서, (원격 클라이언트의 사용자 이름 또는 IP 주소와 같은) 클라이언트별 정보를 로그에 기록하는 것이 바람직 할 수 있습니다. 이를 달성하기 위해 *extra* 매개 변수를 사용할 수는 있지만, 이러한 방식으로 정보를 전달하는 것이 항상 편리하지는 않습니다. 연결마다 `Logger` 인스턴스를 만들고 싶을지 모르지만, 이러한 인스턴스는 가비지 수집되지 않기 때문에 좋지 않습니다. `Logger` 인스턴스의 수가 응용 프로그램 로깅에 사용하고자 하는 세분성 수준에 의존적일 때 이것이 실제로 문제가 되지 않는지만, `Logger` 인스턴스의 수가 실질적으로 무제한이 되면 관리하기 어려울 수 있습니다.

9.1 문맥 정보 전달에 `LoggerAdapters` 사용하기

로깅 이벤트 정보와 함께 출력되는 문맥 정보를 전달하는 쉬운 방법은 `LoggerAdapter` 클래스를 사용하는 것입니다. 이 클래스는 `Logger` 처럼 보이도록 설계되어 있어서, `debug()`, `info()`, `warning()`, `error()`, `exception()`, `critical()` 및 `log()` 를 호출할 수 있습니다. 이 메서드들은 `Logger` 에 있는 것과 똑같은 서명을 가지므로, 두 형의 인스턴스를 같은 의미로 사용할 수 있습니다.

`LoggerAdapter` 의 인스턴스를 생성할 때, `Logger` 인스턴스와 문맥 정보가 포함된 디서너리류 객체를 전달합니다. `LoggerAdapter` 의 인스턴스에서 로깅 메서드 중 하나를 호출하면, 생성자에 전달된 하위 `Logger` 인스턴스에 호출을 위임하고, 이 호출에 문맥 정보를 전달하도록 배치합니다. 다음은 `LoggerAdapter` 코드에서 발췌한 내용입니다:

```
def debug(self, msg, /, *args, **kwargs):
    """
    Delegate a debug call to the underlying logger, after adding
    contextual information from this adapter instance.
    """
    msg, kwargs = self.process(msg, kwargs)
    self.logger.debug(msg, *args, **kwargs)
```

LoggerAdapter 의 process() 메서드는 문맥 정보가 로그 출력에 추가되는 곳입니다. 로깅 호출의 메시지 및 키워드 인자를 받아서, 하부 로거에 대한 호출에서 사용될 (대체로) 수정된 버전을 돌려줍니다. 이 메서드의 기본 구현은 메시지는 그대로 두고, 키워드 인자에 생성자로 전달된 딕셔너리 객체를 값으로 갖는 'extra' 키를 삽입합니다. 물론, 어댑터에 대한 호출에서 'extra' 키워드 인자를 전달한 경우 자동으로 덮어씁니다.

'extra' 를 사용하는 장점은, 딕셔너리 객체에 들어있는 값이 LogRecord 인스턴스의 __dict__ 에 병합되어, 키에 대해 알고 있는 Formatter 인스턴스로 사용자 정의된 문자열을 사용할 수 있게 된다는 것입니다. 다른 방법이 필요한 경우, 가령 메시지 문자열의 앞이나 뒤에 문맥 정보를 덧붙이려는 경우, LoggerAdapter 의 서브 클래스를 만들고, 필요한 작업을 수행하기 위해 process() 를 재정의해야 합니다. 다음은 간단한 예제입니다:

```
class CustomAdapter(logging.LoggerAdapter):
    """
    This example adapter expects the passed in dict-like object to have a
    'connid' key, whose value in brackets is prepended to the log message.
    """
    def process(self, msg, kwargs):
        return '%s] %s' % (self.extra['connid'], msg), kwargs
```

이런 식으로 사용할 수 있습니다:

```
logger = logging.getLogger(__name__)
adapter = CustomAdapter(logger, {'connid': some_conn_id})
```

그러면 어댑터에 로그 하는 모든 이벤트는 로그 메시지 앞에 some_conn_id 값이 붙습니다.

딕셔너리 이외의 객체를 사용하여 문맥 정보 전달하기

실제 딕셔너리를 LoggerAdapter 에 전달할 필요는 없습니다 - 로깅에 딕셔너리처럼 보일 수 있도록, __getitem__ 과 __iter__ 를 구현하는 클래스의 인스턴스를 전달할 수 있습니다. 값을 동적으로 생성하려는 경우 (반면에 딕셔너리에 들어있는 값은 바뀌지 않습니다) 유용합니다.

9.2 문맥 정보 전달에 필터 사용하기

사용자 정의 Filter 를 사용하여 로그 출력에 문맥 정보를 추가할 수도 있습니다. Filter 인스턴스는 전달된 LogRecords 를 수정할 수 있는데, 어트리뷰트를 추가해서 적절한 포맷 문자열이나 필요하다면 사용자 정의 Formatter 를 사용해서 출력되도록 할 수 있습니다.

예를 들어 웹 응용 프로그램에서, 처리 중인 요청(또는 적어도 그것의 흥미로운 부분)을 스레드 로컬 (threading.local) 변수에 저장한 다음, Filter 에서 액세스해서, 요청에서 온 정보를 - 원격 IP 주소와 원격 사용자의 사용자 이름이라고 합시다 - 위의 LoggerAdapter 예제에서와같이 어트리뷰트 이름 'ip' 와 'user' 를 사용하여 LogRecord 에 추가할 수 있습니다. 이 경우 같은 포맷 문자열을 사용하여 위에 표시된 것과 비슷한 출력을 얻을 수 있습니다. 다음은 스크립트 예입니다:

```
import logging
from random import choice

class ContextFilter(logging.Filter):
    """
    This is a filter which injects contextual information into the log.
```

(다음 페이지에 계속)

```

Rather than use actual contextual information, we just use random
data in this demo.
"""

USERS = ['jim', 'fred', 'sheila']
IPS = ['123.231.231.123', '127.0.0.1', '192.168.0.1']

def filter(self, record):

    record.ip = choice(ContextFilter.IPS)
    record.user = choice(ContextFilter.USERS)
    return True

if __name__ == '__main__':
    levels = (logging.DEBUG, logging.INFO, logging.WARNING, logging.ERROR, logging.
↪CRITICAL)
    logging.basicConfig(level=logging.DEBUG,
↪format='% (asctime)-15s %(name)-5s %(levelname)-8s IP:
↪%(ip)-15s User: %(user)-8s %(message)s')
    a1 = logging.getLogger('a.b.c')
    a2 = logging.getLogger('d.e.f')

    f = ContextFilter()
    a1.addFilter(f)
    a2.addFilter(f)
    a1.debug('A debug message')
    a1.info('An info message with %s', 'some parameters')
    for x in range(10):
        lvl = choice(levels)
        lvlname = logging.getLevelName(lvl)
        a2.log(lvl, 'A message at %s level with %d %s', lvlname, 2, 'parameters')

```

실행하면 다음과 같은 결과가 나옵니다:

```

2010-09-06 22:38:15,292 a.b.c DEBUG    IP: 123.231.231.123 User: fred    A debug_
↪message
2010-09-06 22:38:15,300 a.b.c INFO     IP: 192.168.0.1      User: sheila  An info_
↪message with some parameters
2010-09-06 22:38:15,300 d.e.f CRITICAL IP: 127.0.0.1      User: sheila  A_
↪message at CRITICAL level with 2 parameters
2010-09-06 22:38:15,300 d.e.f ERROR    IP: 127.0.0.1      User: jim     A_
↪message at ERROR level with 2 parameters
2010-09-06 22:38:15,300 d.e.f DEBUG    IP: 127.0.0.1      User: sheila  A_
↪message at DEBUG level with 2 parameters
2010-09-06 22:38:15,300 d.e.f ERROR    IP: 123.231.231.123 User: fred    A_
↪message at ERROR level with 2 parameters
2010-09-06 22:38:15,300 d.e.f CRITICAL IP: 192.168.0.1      User: jim     A_
↪message at CRITICAL level with 2 parameters
2010-09-06 22:38:15,300 d.e.f CRITICAL IP: 127.0.0.1      User: sheila  A_
↪message at CRITICAL level with 2 parameters
2010-09-06 22:38:15,300 d.e.f DEBUG    IP: 192.168.0.1      User: jim     A_
↪message at DEBUG level with 2 parameters
2010-09-06 22:38:15,301 d.e.f ERROR    IP: 127.0.0.1      User: sheila  A_
↪message at ERROR level with 2 parameters
2010-09-06 22:38:15,301 d.e.f DEBUG    IP: 123.231.231.123 User: fred    A_

```

(다음 페이지에 계속)

```
↪message at DEBUG level with 2 parameters
2010-09-06 22:38:15,301 d.e.f INFO      IP: 123.231.231.123 User: fred      A_
↪message at INFO level with 2 parameters
```

10 Use of contextvars

Since Python 3.7, the `contextvars` module has provided context-local storage which works for both `threading` and `asyncio` processing needs. This type of storage may thus be generally preferable to `thread-locals`. The following example shows how, in a multi-threaded environment, logs can be populated with contextual information such as, for example, request attributes handled by web applications.

For the purposes of illustration, say that you have different web applications, each independent of the other but running in the same Python process and using a library common to them. How can each of these applications have their own log, where all logging messages from the library (and other request processing code) are directed to the appropriate application's log file, while including in the log additional contextual information such as client IP, HTTP request method and client username?

Let's assume that the library can be simulated by the following code:

```
# webapplib.py
import logging
import time

logger = logging.getLogger(__name__)

def useful():
    # Just a representative event logged from the library
    logger.debug('Hello from webapplib!')
    # Just sleep for a bit so other threads get to run
    time.sleep(0.01)
```

We can simulate the multiple web applications by means of two simple classes, `Request` and `WebApp`. These simulate how real threaded web applications work - each request is handled by a thread:

```
# main.py
import argparse
from contextvars import ContextVar
import logging
import os
from random import choice
import threading
import webapplib

logger = logging.getLogger(__name__)
root = logging.getLogger()
root.setLevel(logging.DEBUG)

class Request:
    """
    A simple dummy request class which just holds dummy HTTP request method,
    client IP address and client username
    """
    def __init__(self, method, ip, user):
        self.method = method
        self.ip = ip
        self.user = user
```

```

# A dummy set of requests which will be used in the simulation - we'll just pick
# from this list randomly. Note that all GET requests are from 192.168.2.XXX
# addresses, whereas POST requests are from 192.16.3.XXX addresses. Three users
# are represented in the sample requests.

REQUESTS = [
    Request('GET', '192.168.2.20', 'jim'),
    Request('POST', '192.168.3.20', 'fred'),
    Request('GET', '192.168.2.21', 'sheila'),
    Request('POST', '192.168.3.21', 'jim'),
    Request('GET', '192.168.2.22', 'fred'),
    Request('POST', '192.168.3.22', 'sheila'),
]

# Note that the format string includes references to request context information
# such as HTTP method, client IP and username

formatter = logging.Formatter('%(threadName)-11s %(appName)s %(name)-9s %(user)-6s
↪ %(ip)s %(method)-4s %(message)s')

# Create our context variables. These will be filled at the start of request
# processing, and used in the logging that happens during that processing

ctx_request = ContextVar('request')
ctx_appname = ContextVar('appname')

class InjectingFilter(logging.Filter):
    """
    A filter which injects context-specific information into logs and ensures
    that only information for a specific webapp is included in its log
    """
    def __init__(self, app):
        self.app = app

    def filter(self, record):
        request = ctx_request.get()
        record.method = request.method
        record.ip = request.ip
        record.user = request.user
        record.appName = appName = ctx_appname.get()
        return appName == self.app.name

class WebApp:
    """
    A dummy web application class which has its own handler and filter for a
    webapp-specific log.
    """
    def __init__(self, name):
        self.name = name
        handler = logging.FileHandler(name + '.log', 'w')
        f = InjectingFilter(self)
        handler.setFormatter(formatter)
        handler.addFilter(f)
        root.addHandler(handler)
        self.num_requests = 0

```

```

def process_request(self, request):
    """
    This is the dummy method for processing a request. It's called on a
    different thread for every request. We store the context information into
    the context vars before doing anything else.
    """
    ctx_request.set(request)
    ctx_appname.set(self.name)
    self.num_requests += 1
    logger.debug('Request processing started')
    webapplib.useful()
    logger.debug('Request processing finished')

def main():
    fn = os.path.splitext(os.path.basename(__file__))[0]
    adhf = argparse.ArgumentDefaultsHelpFormatter
    ap = argparse.ArgumentParser(formatter_class=adhf, prog=fn,
                                description='Simulate a couple of web '
                                            'applications handling some '
                                            'requests, showing how request '
                                            'context can be used to '
                                            'populate logs')

    aa = ap.add_argument
    aa('--count', '-c', type=int, default=100, help='How many requests to simulate
↔')
    options = ap.parse_args()

    # Create the dummy webapps and put them in a list which we can use to select
    # from randomly
    app1 = WebApp('app1')
    app2 = WebApp('app2')
    apps = [app1, app2]
    threads = []
    # Add a common handler which will capture all events
    handler = logging.FileHandler('app.log', 'w')
    handler.setFormatter(formatter)
    root.addHandler(handler)

    # Generate calls to process requests
    for i in range(options.count):
        try:
            # Pick an app at random and a request for it to process
            app = choice(apps)
            request = choice(REQUESTS)
            # Process the request in its own thread
            t = threading.Thread(target=app.process_request, args=(request,))
            threads.append(t)
            t.start()
        except KeyboardInterrupt:
            break

    # Wait for the threads to terminate
    for t in threads:
        t.join()

```

```

for app in apps:
    print('%s processed %s requests' % (app.name, app.num_requests))

if __name__ == '__main__':
    main()

```

If you run the above, you should find that roughly half the requests go into `app1.log` and the rest into `app2.log`, and the all the requests are logged to `app.log`. Each webapp-specific log will contain only log entries for only that webapp, and the request information will be displayed consistently in the log (i.e. the information in each dummy request will always appear together in a log line). This is illustrated by the following shell output:

```

~/logging-contextual-webapp$ python main.py
app1 processed 51 requests
app2 processed 49 requests
~/logging-contextual-webapp$ wc -l *.log
 153 app1.log
 147 app2.log
 300 app.log
 600 total
~/logging-contextual-webapp$ head -3 app1.log
Thread-3 (process_request) app1 __main__ jim 192.168.3.21 POST Request_
↳processing started
Thread-3 (process_request) app1 webapplib jim 192.168.3.21 POST Hello from_
↳webapplib!
Thread-5 (process_request) app1 __main__ jim 192.168.3.21 POST Request_
↳processing started
~/logging-contextual-webapp$ head -3 app2.log
Thread-1 (process_request) app2 __main__ sheila 192.168.2.21 GET Request_
↳processing started
Thread-1 (process_request) app2 webapplib sheila 192.168.2.21 GET Hello from_
↳webapplib!
Thread-2 (process_request) app2 __main__ jim 192.168.2.20 GET Request_
↳processing started
~/logging-contextual-webapp$ head app.log
Thread-1 (process_request) app2 __main__ sheila 192.168.2.21 GET Request_
↳processing started
Thread-1 (process_request) app2 webapplib sheila 192.168.2.21 GET Hello from_
↳webapplib!
Thread-2 (process_request) app2 __main__ jim 192.168.2.20 GET Request_
↳processing started
Thread-3 (process_request) app1 __main__ jim 192.168.3.21 POST Request_
↳processing started
Thread-2 (process_request) app2 webapplib jim 192.168.2.20 GET Hello from_
↳webapplib!
Thread-3 (process_request) app1 webapplib jim 192.168.3.21 POST Hello from_
↳webapplib!
Thread-4 (process_request) app2 __main__ fred 192.168.2.22 GET Request_
↳processing started
Thread-5 (process_request) app1 __main__ jim 192.168.3.21 POST Request_
↳processing started
Thread-4 (process_request) app2 webapplib fred 192.168.2.22 GET Hello from_
↳webapplib!
Thread-6 (process_request) app1 __main__ jim 192.168.3.21 POST Request_
↳processing started
~/logging-contextual-webapp$ grep app1 app1.log | wc -l
153

```

```
~/logging-contextual-webapp$ grep app2 app2.log | wc -l
147
~/logging-contextual-webapp$ grep app1 app.log | wc -l
153
~/logging-contextual-webapp$ grep app2 app.log | wc -l
147
```

11 Imparting contextual information in handlers

Each `Handler` has its own chain of filters. If you want to add contextual information to a `LogRecord` without leaking it to other handlers, you can use a filter that returns a new `LogRecord` instead of modifying it in-place, as shown in the following script:

```
import copy
import logging

def filter(record: logging.LogRecord):
    record = copy.copy(record)
    record.user = 'jim'
    return record

if __name__ == '__main__':
    logger = logging.getLogger()
    logger.setLevel(logging.INFO)
    handler = logging.StreamHandler()
    formatter = logging.Formatter('%(message)s from %(user)-8s')
    handler.setFormatter(formatter)
    handler.addFilter(filter)
    logger.addHandler(handler)

    logger.info('A log message')
```

12 여러 프로세스에서 단일 파일에 로깅 하기

`logging` 이 스레드-안전하고, 단일 프로세스의 여러 스레드에서 단일 파일로 로깅 하는 것이 지원되지만, 파일 안에서 여러 프로세스가 단일 파일에 액세스하는 것을 직렬화하는 표준적인 방법이 없으므로, 여러 프로세스에서 단일 파일로 로깅 하는 것은 지원되지 않습니다. 여러 프로세스에서 하나의 파일에 로그 해야 하는 경우, 이 작업을 수행하는 한 가지 방법은 모든 프로세스가 로그를 `SocketHandler` 에 기록하고, 소켓에서 읽어서 파일로 로그 하는 소켓 서버를 구현하는 별도의 프로세스를 사용하는 것입니다. (원한다면, 기존 프로세스 중 하나에서 한 스레드가 이 기능을 전담하도록 할 수 있습니다.) 이 섹션에서 이 접근법을 더 자세하게 설명하고, 여러분의 응용 프로그램에 적용하기 위한 출발점으로 사용할 수 있는 작동하는 소켓 수신기를 제공합니다.

`multiprocessing` 모듈의 `Lock` 클래스를 사용하는 독자적인 처리기를 작성하여 여러 프로세스에서 파일에 액세스하는 것을 직렬화 할 수 있습니다. 기존 `FileHandler` 와 서브 클래스들은, 앞으로는 가능할 수 있지만, 현재 `multiprocessing` 을 사용하지 않습니다. 현재 `multiprocessing` 모듈이 모든 플랫폼에서 작동하는 록 기능을 제공하지는 않는다는 것에 유의하십시오 (<https://bugs.python.org/issue3770> 를 참조하세요).

또는, `Queue` 와 `QueueHandler` 를 사용하여, 모든 로깅 이벤트를 다중 프로세스 응용 프로그램의 프로세스 중 하나에 보낼 수 있습니다. 다음 예제 스크립트는 이렇게 하는 방법을 보여줍니다; 예제에서 별도의 리스너 프로세스가 다른 프로세스가 보낸 이벤트를 수신하고 자체 로깅 구성에 따라 이벤트를 기록합니다. 이 예제가 한 가지 방법만을 보여 주지만(예를 들어, 별도의 리스너 프로세스 대신 리스너 스레드를 사용할 수도 있습니다 - 구현은 비슷할 것입니다), 리스너와 응용 프로그램의 다른 프로세스가 완전히 다른 로깅 구성을 사용하도록 허용하고, 여러분 자신의 특별한 요구 사항을 충족하는 코드의 기초로 사용할 수 있습니다:

```

# You'll need these imports in your own code
import logging
import logging.handlers
import multiprocessing

# Next two import lines for this demo only
from random import choice, random
import time

#
# Because you'll want to define the logging configurations for listener and
↳workers, the
# listener and worker process functions take a configurer parameter which is a
↳callable
# for configuring logging for that process. These functions are also passed the
↳queue,
# which they use for communication.
#
# In practice, you can configure the listener however you want, but note that in
↳this
# simple example, the listener does not apply level or filter logic to received
↳records.
# In practice, you would probably want to do this logic in the worker processes,
↳to avoid
# sending events which would be filtered out between processes.
#
# The size of the rotated files is made small so you can see the results easily.
def listener_configurer():
    root = logging.getLogger()
    h = logging.handlers.RotatingFileHandler('mptest.log', 'a', 300, 10)
    f = logging.Formatter('%(asctime)s %(processName)-10s %(name)s %(levelname)-8s
↳%(message)s')
    h.setFormatter(f)
    root.addHandler(h)

# This is the listener process top-level loop: wait for logging events
# (LogRecords) on the queue and handle them, quit when you get a None for a
# LogRecord.
def listener_process(queue, configurer):
    configurer()
    while True:
        try:
            record = queue.get()
            if record is None: # We send this as a sentinel to tell the listener
↳to quit.
                break
            logger = logging.getLogger(record.name)
            logger.handle(record) # No level or filter logic applied - just do it!
        except Exception:
            import sys, traceback
            print('Whoops! Problem:', file=sys.stderr)
            traceback.print_exc(file=sys.stderr)

# Arrays used for random selections in this demo
LEVELS = [logging.DEBUG, logging.INFO, logging.WARNING,
           logging.ERROR, logging.CRITICAL]

```

(다음 페이지에 계속)

```

LOGGERS = ['a.b.c', 'd.e.f']

MESSAGES = [
    'Random message #1',
    'Random message #2',
    'Random message #3',
]

# The worker configuration is done at the start of the worker process run.
# Note that on Windows you can't rely on fork semantics, so each process
# will run the logging configuration code when it starts.
def worker_configurer(queue):
    h = logging.handlers.QueueHandler(queue) # Just the one handler needed
    root = logging.getLogger()
    root.addHandler(h)
    # send all messages, for demo; no other level or filter logic applied.
    root.setLevel(logging.DEBUG)

# This is the worker process top-level loop, which just logs ten events with
# random intervening delays before terminating.
# The print messages are just so you know it's doing something!
def worker_process(queue, configurer):
    configurer(queue)
    name = multiprocessing.current_process().name
    print('Worker started: %s' % name)
    for i in range(10):
        time.sleep(random())
        logger = logging.getLogger(choice(LOGGERS))
        level = choice(LEVELS)
        message = choice(MESSAGES)
        logger.log(level, message)
    print('Worker finished: %s' % name)

# Here's where the demo gets orchestrated. Create the queue, create and start
# the listener, create ten workers and start them, wait for them to finish,
# then send a None to the queue to tell the listener to finish.
def main():
    queue = multiprocessing.Queue(-1)
    listener = multiprocessing.Process(target=listener_process,
                                     args=(queue, listener_configurer))

    listener.start()
    workers = []
    for i in range(10):
        worker = multiprocessing.Process(target=worker_process,
                                       args=(queue, worker_configurer))

        workers.append(worker)
        worker.start()
    for w in workers:
        w.join()
    queue.put_nowait(None)
    listener.join()

if __name__ == '__main__':
    main()

```

위의 스크립트 변형은 로깅을 메인 프로세스의 별도의 스레드에서 유지합니다:

```

import logging
import logging.config
import logging.handlers
from multiprocessing import Process, Queue
import random
import threading
import time

def logger_thread(q):
    while True:
        record = q.get()
        if record is None:
            break
        logger = logging.getLogger(record.name)
        logger.handle(record)

def worker_process(q):
    qh = logging.handlers.QueueHandler(q)
    root = logging.getLogger()
    root.setLevel(logging.DEBUG)
    root.addHandler(qh)
    levels = [logging.DEBUG, logging.INFO, logging.WARNING, logging.ERROR,
              logging.CRITICAL]
    loggers = ['foo', 'foo.bar', 'foo.bar.baz',
               'spam', 'spam.ham', 'spam.ham.eggs']
    for i in range(100):
        lvl = random.choice(levels)
        logger = logging.getLogger(random.choice(loggers))
        logger.log(lvl, 'Message no. %d', i)

if __name__ == '__main__':
    q = Queue()
    d = {
        'version': 1,
        'formatters': {
            'detailed': {
                'class': 'logging.Formatter',
                'format': '%(asctime)s %(name)-15s %(levelname)-8s %(processName)-
↪10s %(message)s'
            }
        },
        'handlers': {
            'console': {
                'class': 'logging.StreamHandler',
                'level': 'INFO',
            },
            'file': {
                'class': 'logging.FileHandler',
                'filename': 'mplog.log',
                'mode': 'w',
                'formatter': 'detailed',
            },
            'foofile': {
                'class': 'logging.FileHandler',
                'filename': 'mplog-foo.log',
                'mode': 'w',
            }
        }
    }

```

(다음 페이지에 계속)

```

        'formatter': 'detailed',
    },
    'errors': {
        'class': 'logging.FileHandler',
        'filename': 'mplog-errors.log',
        'mode': 'w',
        'level': 'ERROR',
        'formatter': 'detailed',
    },
},
'loggers': {
    'foo': {
        'handlers': ['foofile']
    }
},
'root': {
    'level': 'DEBUG',
    'handlers': ['console', 'file', 'errors']
},
}
workers = []
for i in range(5):
    wp = Process(target=worker_process, name='worker %d' % (i + 1), args=(q,))
    workers.append(wp)
    wp.start()
logging.config.dictConfig(d)
lp = threading.Thread(target=logger_thread, args=(q,))
lp.start()
# At this point, the main process could do some useful work of its own
# Once it's done that, it can wait for the workers to terminate...
for wp in workers:
    wp.join()
# And now tell the logging thread to finish up, too
q.put(None)
lp.join()

```

이 변형은 특정 로거에 대한 구성을 적용하는 방법을 보여줍니다. 예를 들어, foo 로거는 foo 서버 시스템의 모든 이벤트를 mplog-foo.log 파일에 저장하는 특별한 처리기를 갖고 있습니다. 이것은 메인 프로세스의 로깅 시스템이 (로깅 이벤트가 작업자 프로세스에서 만들어졌다 하더라도) 메시지를 적절한 대상으로 전달하는 데 사용됩니다.

12.1 concurrent.futures.ProcessPoolExecutor 사용하기

concurrent.futures.ProcessPoolExecutor를 사용하여 작업자 프로세스를 시작하려면, 약간 다른 방법으로 큐를 만들어야 합니다.

```
queue = multiprocessing.Queue(-1)
```

대신에, 다음을 사용해야 합니다

```
queue = multiprocessing.Manager().Queue(-1) # also works with the examples above
```

그런 다음 작업자 생성을:

```
workers = []
for i in range(10):
    worker = multiprocessing.Process(target=worker_process,
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
args=(queue, worker_configurer))

workers.append(worker)
worker.start()
for w in workers:
    w.join()
```

에서 다음과 같이 대체할 수 있습니다 (먼저 `concurrent.futures`를 임포트 하는 것을 기억하십시오):

```
with concurrent.futures.ProcessPoolExecutor(max_workers=10) as executor:
    for i in range(10):
        executor.submit(worker_process, queue, worker_configurer)
```

12.2 Deploying Web applications using Gunicorn and uWSGI

When deploying Web applications using [Gunicorn](#) or [uWSGI](#) (or similar), multiple worker processes are created to handle client requests. In such environments, avoid creating file-based handlers directly in your web application. Instead, use a `SocketHandler` to log from the web application to a listener in a separate process. This can be set up using a process management tool such as Supervisor - see [Running a logging socket listener in production](#) for more details.

13 파일 회전 사용하기

Sometimes you want to let a log file grow to a certain size, then open a new file and log to that. You may want to keep a certain number of these files, and when that many files have been created, rotate the files so that the number of files and the size of the files both remain bounded. For this usage pattern, the logging package provides a `RotatingFileHandler`:

```
import glob
import logging
import logging.handlers

LOG_FILENAME = 'logging_rotatingfile_example.out'

# Set up a specific logger with our desired output level
my_logger = logging.getLogger('MyLogger')
my_logger.setLevel(logging.DEBUG)

# Add the log message handler to the logger
handler = logging.handlers.RotatingFileHandler(
    LOG_FILENAME, maxBytes=20, backupCount=5)

my_logger.addHandler(handler)

# Log some messages
for i in range(20):
    my_logger.debug('i = %d' % i)

# See what files are created
logfiles = glob.glob('%s*' % LOG_FILENAME)

for filename in logfiles:
    print(filename)
```

결과는 6개의 파일이어야 하고, 각기 응용 프로그램에 대한 로그 기록의 일부입니다:

```
logging_rotatingfile_example.out
logging_rotatingfile_example.out.1
logging_rotatingfile_example.out.2
logging_rotatingfile_example.out.3
logging_rotatingfile_example.out.4
logging_rotatingfile_example.out.5
```

가장 최근의 파일은 항상 `logging_rotatingfile_example.out` 이며, 크기 제한에 도달할 때마다 접미사 `.1` 이 붙은 이름으로 변경됩니다. 기존 백업 파일 각각의 이름이 변경되어 접미사가 증가하고 (`.1` 이 `.2` 가 되는 등) `.6` 파일이 지워집니다.

분명, 이 예제는 로그 길이를 극단적으로 작게 설정합니다. `maxBytes` 를 적절한 값으로 설정하고 싶을 겁니다.

14 대체 포매팅 스타일 사용하기

`logging` 이 파이썬 표준 라이브러리에 추가되었을 때, 가변 내용으로 메시지를 포맷하는 유일한 방법은 %-포매팅 방법을 사용하는 것이었습니다. 그 이후로, 파이썬은 두 개의 새로운 포매팅 접근법을 얻었습니다: `string.Template` (파이썬 2.4에 추가됨)과 `str.format()` (파이썬 2.6에 추가됨).

로깅은 (3.2부터) 이 두 가지 추가 포매팅 스타일에 대해 개선된 지원을 제공합니다. `Formatter` 클래스는 `style` 이라는 추가적인 키워드 매개 변수를 취하도록 개선되었습니다. 기본값은 `'%'` 이지만, 다른 두 가지 포매팅 스타일에 해당하는 `'{'` 및 `'$'` 를 사용할 수 있습니다. (여러분이 기대하듯이) 이전 버전과의 호환성은 기본적으로 유지되지만, `style` 매개 변수를 명시적으로 지정하면 `str.format()` 또는 `string.Template` 과 함께 작동하는 포맷 문자열을 지정할 수 있습니다. 다음은 가능성을 보여주기 위한 예제 콘솔 세션입니다:

```
>>> import logging
>>> root = logging.getLogger()
>>> root.setLevel(logging.DEBUG)
>>> handler = logging.StreamHandler()
>>> bf = logging.Formatter('{asctime} {name} {levelname:8s} {message}',
...                         style='{')
>>> handler.setFormatter(bf)
>>> root.addHandler(handler)
>>> logger = logging.getLogger('foo.bar')
>>> logger.debug('This is a DEBUG message')
2010-10-28 15:11:55,341 foo.bar DEBUG      This is a DEBUG message
>>> logger.critical('This is a CRITICAL message')
2010-10-28 15:12:11,526 foo.bar CRITICAL This is a CRITICAL message
>>> df = logging.Formatter('$asctime $name ${levelname} $message',
...                         style='$')
>>> handler.setFormatter(df)
>>> logger.debug('This is a DEBUG message')
2010-10-28 15:13:06,924 foo.bar DEBUG This is a DEBUG message
>>> logger.critical('This is a CRITICAL message')
2010-10-28 15:13:11,494 foo.bar CRITICAL This is a CRITICAL message
>>>
```

로그로 최종 출력하기 위해 로깅 메시지를 포맷하는 것은 개별 로깅 메시지가 구성되는 방식과 완전히 별개입니다. 개별 메시지에는 다음과 같이 %-포매팅을 사용할 수 있습니다:

```
>>> logger.error('This is an%s %s %s', 'other,', 'ERROR,', 'message')
2010-10-28 15:19:29,833 foo.bar ERROR This is another, ERROR, message
>>>
```

로깅 호출(`logger.debug()`, `logger.info()` 등)은 실제 로깅 메시지 자체를 위해서는 위치 매개 변수만을 취하고, 키워드 매개 변수는 실제 로깅 호출을 어떻게 다뤄야 하는지를 지정하는 옵션을 결정하는

용도로만 사용됩니다(예를 들어, 트레이스백 정보를 로그 해야 할지를 가리키는 `exc_info` 키워드 매개 변수나 로그에 추가되는 문맥 정보를 나타내는 `extra` 키워드 매개 변수). 그래서 여러분은 `str.format()` 또는 `string.Template` 문법을 사용하여 직접 로깅 호출을 할 수 없습니다, 내부적으로 `logging` 패키지가 %-포매팅을 사용하여 포맷 문자열과 변수 인자를 병합하기 때문입니다. 이전 버전과의 호환성을 유지하는 동안은 이 상황이 바뀌지 않을 것입니다. 기존 코드에 있는 모든 로깅 호출이 %-포맷 문자열을 사용하기 때문입니다.

그러나 {}- 및 \$- 포매팅을 사용하여 개별 로그 메시지를 구성하는 방법이 있습니다. 메시지의 경우, 메시지 포맷 문자열로 임의의 객체를 사용할 수 있으며, `logging` 패키지는 실제 형식 문자열을 얻기 위해 그 객체에 대해 `str()` 을 호출한다는 것을 상기하십시오. 다음 두 가지 클래스를 고려하십시오:

```
class BraceMessage:
    def __init__(self, fmt, /, *args, **kwargs):
        self.fmt = fmt
        self.args = args
        self.kwargs = kwargs

    def __str__(self):
        return self.fmt.format(*self.args, **self.kwargs)

class DollarMessage:
    def __init__(self, fmt, /, **kwargs):
        self.fmt = fmt
        self.kwargs = kwargs

    def __str__(self):
        from string import Template
        return Template(self.fmt).substitute(**self.kwargs)
```

이 중 하나를 포맷 문자열 대신 사용하면, {}- 또는 \$-포매팅을 사용하여 포맷된 로그 출력의 “%(message)s”, “[message]” 또는 “\$message” 자리에 나타나는 실제 “message” 부분을 만들 수 있습니다. 어떤 것을 로그 하고 싶을 때마다 클래스 이름을 사용하는 것은 다소 낱사납지만, __ (두 개의 밑줄 --- `gettext.gettext()`) 나 그 형제들의 동의어/별칭으로 사용되는 _ 과 혼동하지 마세요)와 같은 별칭을 사용하면 꽤 쓸만합니다.

위의 클래스가 파이썬에 포함되어 있지는 않지만, 아주 쉽게 여러분의 코드에 복사하여 붙여넣을 수 있습니다. 다음과 같이 사용될 수 있습니다 (wherever 라는 모듈에서 선언되었다고 가정합니다):

```
>>> from wherever import BraceMessage as __
>>> print(__('Message with {0} {name}', 2, name='placeholders'))
Message with 2 placeholders
>>> class Point: pass
...
>>> p = Point()
>>> p.x = 0.5
>>> p.y = 0.5
>>> print(__('Message with coordinates: ({point.x:.2f}, {point.y:.2f})',
...         point=p))
Message with coordinates: (0.50, 0.50)
>>> from wherever import DollarMessage as __
>>> print(__('Message with $num $what', num=2, what='placeholders'))
Message with 2 placeholders
>>>
```

위의 예제는 `print()` 를 사용하여 포매팅이 어떻게 작동하는지 보여주고 있지만, 물론 이 접근법으로 실제 로깅 할 때는 `logger.debug()` 나 그와 유사한 것을 사용해야 합니다.

One thing to note is that you pay no significant performance penalty with this approach: the actual formatting happens not when you make the logging call, but when (and if) the logged message is actually about to be output to a log by a handler. So the only slightly unusual thing which might trip you up is that the parentheses go around the format string and the arguments, not just the format string. That’s because the __ notation is just syntax sugar for a constructor

call to one of the XXXMessage classes.

원한다면, LoggerAdapter 를 사용하여 다음 예제와 같이 위와 유사한 효과를 얻을 수 있습니다:

```
import logging

class Message:
    def __init__(self, fmt, args):
        self.fmt = fmt
        self.args = args

    def __str__(self):
        return self.fmt.format(*self.args)

class StyleAdapter(logging.LoggerAdapter):
    def log(self, level, msg, /, *args, stacklevel=1, **kwargs):
        if self.isEnabledFor(level):
            msg, kwargs = self.process(msg, kwargs)
            self.logger.log(level, Message(msg, args), **kwargs,
                           stacklevel=stacklevel+1)

logger = StyleAdapter(logging.getLogger(__name__))

def main():
    logger.debug('Hello, {}', 'world!')

if __name__ == '__main__':
    logging.basicConfig(level=logging.DEBUG)
    main()
```

The above script should log the message Hello, world! when run with Python 3.8 or later.

15 사용자 정의 LogRecord

모든 로깅 이벤트는 LogRecord 인스턴스로 표현됩니다. 이벤트가 로그되고 로거 수준에 의해 필터링 되지 않으면, LogRecord 가 생성되고 이벤트에 대한 정보로 채워진 다음 해당 로거(와 그 조상들, 계층 상위로의 전파가 비활성화된 지점의 로거까지)의 처리기로 전달됩니다. 파이썬 3.2 이전에는, 이 생성이 일어나는 곳이 두 곳밖에 없었습니다:

- `Logger.makeRecord()`, 이벤트 로깅의 일반적인 프로세스에서 호출됩니다. 인스턴스를 생성하기 위해 `LogRecord` 를 직접 호출합니다.
- `makeLogRecord()`, `LogRecord` 에 추가될 어트리뷰트를 포함하는 디셔너리와 함께 호출됩니다. 보통 적절한 디셔너리가 네트워크를 통해 (예를 들어, `SocketHandler` 를 통해 피클 형태로, 또는 `HTTPHandler` 를 통해 JSON 형식으로) 수신될 때 호출됩니다.

이것은 보통 `LogRecord` 로 특별한 것을 할 필요가 있다면, 다음 중 하나를 해야 한다는 것을 의미합니다.

- `Logger.makeRecord()` 를 재정의하는 자신만의 `Logger` 서브 클래스를 만들고, 관심 있는 로거의 인스턴스가 만들어지기 전에 `setLoggerClass()` 를 사용하여 설정하십시오.
- 로거나 처리기에 `Filter` 를 추가해서 `filter()` 메서드가 호출될 때 필요한 특별한 조작을 하십시오.

첫 번째 접근법은 여러 라이브러리가 서로 다른 일을 하고 싶어 하는 시나리오에서는 다루기 힘들 것입니다. 각자 자신의 `Logger` 서브 클래스를 설정하려고 시도할 것이고, 마지막 것이 이기게 될 것입니다.

두 번째 접근법은 많은 경우에 합리적으로 잘 작동하지만, `LogRecord` 의 특별한 서브 클래스를 사용할 수는 없습니다. 라이브러리 개발자는 로거에 적절한 필터를 설정할 수 있지만, 새로운 로거를 도입할 때마다 이를 수행해야 한다는 것을 기억해야 합니다. 이를 고려하지 않는다면 새 패키지나 모듈을 추가하고 모듈 수준에서 단순히 다음과 같이 합니다:

```
logger = logging.getLogger(__name__)
```

이것은 아마도 고려해야 할 많은 것 중 하나일 뿐입니다. 개발자는 자신의 최상위 로거에 첨부된 `NullHandler` 에도 필터를 추가 할 수 있지만, 응용 프로그램 개발자가 하위 수준 라이브러리 로거에 처리기를 연결하면 호출되지 않습니다 — 그래서 그 처리기로부터의 출력은 라이브러리 개발자의 의도를 반영하지 못합니다.

파이썬 3.2 이상에서는, `LogRecord` 생성이 사용자가 지정할 수 있는 팩토리를 통해 수행됩니다. 팩토리는 `setLogRecordFactory()` 로 설정할 수 있고, `getLogRecordFactory()` 로 조회할 수 있는 콜러블입니다. 팩토리는 `LogRecord` 생성자와 같은 서명으로 호출되고, 기본 설정은 `LogRecord` 입니다.

이 방법을 사용하면 사용자 정의 팩토리가 `LogRecord` 생성의 모든 측면을 제어 할 수 있습니다. 예를 들어, 서브 클래스를 반환하거나, 다음과 같은 방법으로 생성된 레코드에 어트리뷰트를 추가 할 수 있습니다:

```
old_factory = logging.getLogRecordFactory()

def record_factory(*args, **kwargs):
    record = old_factory(*args, **kwargs)
    record.custom_attribute = 0xdecafbad
    return record

logging.setLogRecordFactory(record_factory)
```

이 패턴은 서로 다른 라이브러리가 팩토리를 체인으로 연결할 수 있도록 하며, 서로의 어트리뷰트를 덮어 쓰거나 의도하지 않게 표준으로 제공된 어트리뷰트를 덮어쓰지 않는 한, 놀랄 일은 없어야 합니다. 그러나 체인의 각 고리는 모든 로깅 작업에 실행시간 오버헤드를 추가하므로, `Filter` 를 사용해서 원하는 결과를 얻을 수 없을 때만 이 기법을 사용해야 합니다.

16 Subclassing QueueHandler and QueueListener- a ZeroMQ example

16.1 Subclass QueueHandler

`QueueHandler` 서브 클래스를 사용하여 다른 유형의 큐에 메시지를 보낼 수 있습니다, 예를 들어 `ZeroMQ` 'publish' 소켓. 아래 예제에서, 소켓은 별도로 생성되어 처리기로 ('queue' 로) 전달됩니다:

```
import zmq # using pyzmq, the Python binding for ZeroMQ
import json # for serializing records portably

ctx = zmq.Context()
sock = zmq.Socket(ctx, zmq.PUB) # or zmq.PUSH, or other suitable value
sock.bind('tcp://*:5556') # or wherever

class ZeroMQSocketHandler(QueueHandler):
    def enqueue(self, record):
        self.queue.send_json(record.__dict__)

handler = ZeroMQSocketHandler(sock)
```

물론 구성하는 다른 방법이 있습니다. 예를 들어 처리기가 소켓을 만드는데 필요한 데이터를 전달하는 것입니다:

```
class ZeroMQSocketHandler(QueueHandler):
    def __init__(self, uri, socktype=zmq.PUB, ctx=None):
        self.ctx = ctx or zmq.Context()
        socket = zmq.Socket(self.ctx, socktype)
```

(다음 페이지에 계속)

```

socket.bind(uri)
super().__init__(socket)

def enqueue(self, record):
    self.queue.send_json(record.__dict__)

def close(self):
    self.queue.close()

```

16.2 Subclass QueueListener

다른 유형의 큐에서 메시지를 받기 위해 QueueListener 의 서브 클래스를 만들 수도 있습니다, 예를 들어 ZeroMQ ‘subscribe’ 소켓. 다음은 그 예입니다:

```

class ZeroMQSocketListener(QueueListener):
    def __init__(self, uri, /, *handlers, **kwargs):
        self.ctx = kwargs.get('ctx') or zmq.Context()
        socket = zmq.Socket(self.ctx, zmq.SUB)
        socket.setsockopt_string(zmq.SUBSCRIBE, '') # subscribe to everything
        socket.connect(uri)
        super().__init__(socket, *handlers, **kwargs)

    def dequeue(self):
        msg = self.queue.recv_json()
        return logging.makeLogRecord(msg)

```

17 Subclassing QueueHandler and QueueListener- a pyngg example

In a similar way to the above section, we can implement a listener and handler using pyngg, which is a Python binding to NNG, billed as a spiritual successor to ZeroMQ. The following snippets illustrate – you can test them in an environment which has pyngg installed. Just for variety, we present the listener first.

17.1 Subclass QueueListener

```

# listener.py
import json
import logging
import logging.handlers

import pyngg

DEFAULT_ADDR = "tcp://localhost:13232"

interrupted = False

class NNGSocketListener(logging.handlers.QueueListener):

    def __init__(self, uri, /, *handlers, **kwargs):
        # Have a timeout for interruptability, and open a
        # subscriber socket
        socket = pyngg.Sub0(listen=uri, recv_timeout=500)
        # The b'' subscription matches all topics
        topics = kwargs.pop('topics', None) or b''

```

```

socket.subscribe(topics)
# We treat the socket as a queue
super().__init__(socket, *handlers, **kwargs)

def dequeue(self, block):
    data = None
    # Keep looping while not interrupted and no data received over the
    # socket
    while not interrupted:
        try:
            data = self.queue.recv(block=block)
            break
        except pynng.Timeout:
            pass
        except pynng.Closed: # sometimes happens when you hit Ctrl-C
            break
    if data is None:
        return None
    # Get the logging event sent from a publisher
    event = json.loads(data.decode('utf-8'))
    return logging.makeLogRecord(event)

def enqueue_sentinel(self):
    # Not used in this implementation, as the socket isn't really a
    # queue
    pass

logging.getLogger('pynng').propagate = False
listener = NNGSocketListener(DEFAULT_ADDR, logging.StreamHandler(), topics=b'')
listener.start()
print('Press Ctrl-C to stop.')
try:
    while True:
        pass
except KeyboardInterrupt:
    interrupted = True
finally:
    listener.stop()

```

17.2 Subclass QueueHandler

```

# sender.py
import json
import logging
import logging.handlers
import time
import random

import pynng

DEFAULT_ADDR = "tcp://localhost:13232"

class NNGSocketHandler(logging.handlers.QueueHandler):

    def __init__(self, uri):

```

```

socket = pynng.Pub0(dial=uri, send_timeout=500)
super().__init__(socket)

def enqueue(self, record):
    # Send the record as UTF-8 encoded JSON
    d = dict(record.__dict__)
    data = json.dumps(d)
    self.queue.send(data.encode('utf-8'))

def close(self):
    self.queue.close()

logging.getLogger('pynng').propagate = False
handler = NNGSocketHandler(DEFAULT_ADDR)
# Make sure the process ID is in the output
logging.basicConfig(level=logging.DEBUG,
                    handlers=[logging.StreamHandler(), handler],
                    format='%(levelname)-8s %(name)10s %(process)6s %(message)s')
levels = (logging.DEBUG, logging.INFO, logging.WARNING, logging.ERROR,
          logging.CRITICAL)
logger_names = ('myapp', 'myapp.lib1', 'myapp.lib2')
msgno = 1
while True:
    # Just randomly select some loggers and levels and log away
    level = random.choice(levels)
    logger = logging.getLogger(random.choice(logger_names))
    logger.log(level, 'Message no. %5d' % msgno)
    msgno += 1
    delay = random.random() * 2 + 0.5
    time.sleep(delay)

```

You can run the above two snippets in separate command shells. If we run the listener in one shell and run the sender in two separate shells, we should see something like the following. In the first sender shell:

```

$ python sender.py
DEBUG      myapp      613 Message no.      1
WARNING    myapp.lib2   613 Message no.      2
CRITICAL   myapp.lib2   613 Message no.      3
WARNING    myapp.lib2   613 Message no.      4
CRITICAL   myapp.lib1   613 Message no.      5
DEBUG      myapp      613 Message no.      6
CRITICAL   myapp.lib1   613 Message no.      7
INFO       myapp.lib1   613 Message no.      8

```

(and so on)

In the second sender shell:

```

$ python sender.py
INFO       myapp.lib2   657 Message no.      1
CRITICAL   myapp.lib2   657 Message no.      2
CRITICAL   myapp      657 Message no.      3
CRITICAL   myapp.lib1   657 Message no.      4
INFO       myapp.lib1   657 Message no.      5
WARNING    myapp.lib2   657 Message no.      6
CRITICAL   myapp      657 Message no.      7
DEBUG      myapp.lib1   657 Message no.      8

```

(and so on)

In the listener shell:

```
$ python listener.py
Press Ctrl-C to stop.
DEBUG          myapp      613 Message no.      1
WARNING        myapp.lib2  613 Message no.      2
INFO           myapp.lib2  657 Message no.      1
CRITICAL       myapp.lib2  613 Message no.      3
CRITICAL       myapp.lib2  657 Message no.      2
CRITICAL       myapp      657 Message no.      3
WARNING        myapp.lib2  613 Message no.      4
CRITICAL       myapp.lib1  613 Message no.      5
CRITICAL       myapp.lib1  657 Message no.      4
INFO           myapp.lib1  657 Message no.      5
DEBUG          myapp      613 Message no.      6
WARNING        myapp.lib2  657 Message no.      6
CRITICAL       myapp      657 Message no.      7
CRITICAL       myapp.lib1  613 Message no.      7
INFO           myapp.lib1  613 Message no.      8
DEBUG          myapp.lib1  657 Message no.      8
(and so on)
```

As you can see, the logging from the two sender processes is interleaved in the listener's output.

18 딕셔너리 기반 구성의 예

다음은 로깅 구성 딕셔너리의 예입니다 - 장고 프로젝트 설명서 에서 가져왔습니다. 이 딕셔너리를 dictConfig() 로 전달하여 구성을 적용합니다:

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'verbose': {
            'format': '{levelname} {asctime} {module} {process:d} {thread:d}
↪{message}',
            'style': '{',
        },
        'simple': {
            'format': '{levelname} {message}',
            'style': '{',
        },
    },
    'filters': {
        'special': {
            '()': 'project.logging.SpecialFilter',
            'foo': 'bar',
        },
    },
    'handlers': {
        'console': {
            'level': 'INFO',
            'class': 'logging.StreamHandler',
            'formatter': 'simple',
        },
        'mail_admins': {
            'level': 'ERROR',
```

(다음 페이지에 계속)

```

        'class': 'django.utils.log.AdminEmailHandler',
        'filters': ['special']
    },
},
'loggers': {
    'django': {
        'handlers': ['console'],
        'propagate': True,
    },
    'django.request': {
        'handlers': ['mail_admins'],
        'level': 'ERROR',
        'propagate': False,
    },
    'myproject.custom': {
        'handlers': ['console', 'mail_admins'],
        'level': 'INFO',
        'filters': ['special']
    }
}
}

```

이 구성에 대한 더 자세한 정보는 [장고 설명서의 관련 섹션](#) 을 참조하세요.

19 rotator와 namer를 사용해서 로그 회전 처리하기

An example of how you can define a namer and rotator is given in the following runnable script, which shows gzip compression of the log file:

```

import gzip
import logging
import logging.handlers
import os
import shutil

def namer(name):
    return name + ".gz"

def rotator(source, dest):
    with open(source, 'rb') as f_in:
        with gzip.open(dest, 'wb') as f_out:
            shutil.copyfileobj(f_in, f_out)
    os.remove(source)

rh = logging.handlers.RotatingFileHandler('rotated.log', maxBytes=128,
↳backupCount=5)
rh.rotator = rotator
rh.namer = namer

root = logging.getLogger()
root.setLevel(logging.INFO)
root.addHandler(rh)
f = logging.Formatter('%(asctime)s %(message)s')
rh.setFormatter(f)
for i in range(1000):

```

```
root.info(f'Message no. {i + 1}')
```

After running this, you will see six new files, five of which are compressed:

```
$ ls rotated.log*
rotated.log      rotated.log.2.gz  rotated.log.4.gz
rotated.log.1.gz rotated.log.3.gz  rotated.log.5.gz
$ zcat rotated.log.1.gz
2023-01-20 02:28:17,767 Message no. 996
2023-01-20 02:28:17,767 Message no. 997
2023-01-20 02:28:17,767 Message no. 998
```

20 좀 더 정교한 multiprocessing 예제

다음 동작하는 예제에서는 구성 파일을 사용하여 로깅을 multiprocessing과 함께 사용하는 방법을 보여줍니다. 구성은 매우 간단하지만, 실제 multiprocessing 시나리오에서 더 복잡한 구성을 구현할 수 있음을 예시합니다.

이 예에서, 주 프로세스는 리스너 프로세스와 몇 개의 작업자 프로세스를 생성합니다. 주 프로세스, 리스너 및 작업자를 위한 세 가지 구성이 있습니다 (모든 작업자는 같은 구성을 공유합니다). 주 프로세스에서의 로깅, 작업자가 QueueHandler에 로그 하는 방법, 그리고 리스너가 QueueListener 및 더욱 복잡한 로깅 구성을 구현하고 큐를 통해 수신한 이벤트를 구성에서 지정된 처리기로 전달하도록 배치하는 방법을 볼 수 있습니다. 이러한 구성은 설명을 위한 것이지만, 이 예제를 여러분 자신의 시나리오에 적용할 수 있어야 합니다.

스크립트는 다음과 같습니다 - 독스트링과 주석이 어떻게 작동하는지 잘 설명하기를 바랍니다:

```
import logging
import logging.config
import logging.handlers
from multiprocessing import Process, Queue, Event, current_process
import os
import random
import time

class MyHandler:
    """
    A simple handler for logging events. It runs in the listener process and
    dispatches events to loggers based on the name in the received record,
    which then get dispatched, by the logging system, to the handlers
    configured for those loggers.
    """

    def handle(self, record):
        if record.name == "root":
            logger = logging.getLogger()
        else:
            logger = logging.getLogger(record.name)

        if logger.isEnabledFor(record.levelno):
            # The process name is transformed just to show that it's the listener
            # doing the logging to files and console
            record.processName = '%s (for %s)' % (current_process().name, record.
↵processName)
            logger.handle(record)
```

```

def listener_process(q, stop_event, config):
    """
    This could be done in the main process, but is just done in a separate
    process for illustrative purposes.

    This initialises logging according to the specified configuration,
    starts the listener and waits for the main process to signal completion
    via the event. The listener is then stopped, and the process exits.
    """
    logging.config.dictConfig(config)
    listener = logging.handlers.QueueListener(q, MyHandler())
    listener.start()
    if os.name == 'posix':
        # On POSIX, the setup logger will have been configured in the
        # parent process, but should have been disabled following the
        # dictConfig call.
        # On Windows, since fork isn't used, the setup logger won't
        # exist in the child, so it would be created and the message
        # would appear - hence the "if posix" clause.
        logger = logging.getLogger('setup')
        logger.critical('Should not appear, because of disabled logger ...')
    stop_event.wait()
    listener.stop()

def worker_process(config):
    """
    A number of these are spawned for the purpose of illustration. In
    practice, they could be a heterogeneous bunch of processes rather than
    ones which are identical to each other.

    This initialises logging according to the specified configuration,
    and logs a hundred messages with random levels to randomly selected
    loggers.

    A small sleep is added to allow other processes a chance to run. This
    is not strictly needed, but it mixes the output from the different
    processes a bit more than if it's left out.
    """
    logging.config.dictConfig(config)
    levels = [logging.DEBUG, logging.INFO, logging.WARNING, logging.ERROR,
              logging.CRITICAL]
    loggers = ['foo', 'foo.bar', 'foo.bar.baz',
               'spam', 'spam.ham', 'spam.ham.eggs']
    if os.name == 'posix':
        # On POSIX, the setup logger will have been configured in the
        # parent process, but should have been disabled following the
        # dictConfig call.
        # On Windows, since fork isn't used, the setup logger won't
        # exist in the child, so it would be created and the message
        # would appear - hence the "if posix" clause.
        logger = logging.getLogger('setup')
        logger.critical('Should not appear, because of disabled logger ...')
    for i in range(100):
        lvl = random.choice(levels)
        logger = logging.getLogger(random.choice(loggers))
        logger.log(lvl, 'Message no. %d', i)

```

```

time.sleep(0.01)

def main():
    q = Queue()
    # The main process gets a simple configuration which prints to the console.
    config_initial = {
        'version': 1,
        'handlers': {
            'console': {
                'class': 'logging.StreamHandler',
                'level': 'INFO'
            }
        },
        'root': {
            'handlers': ['console'],
            'level': 'DEBUG'
        }
    }
    # The worker process configuration is just a QueueHandler attached to the
    # root logger, which allows all messages to be sent to the queue.
    # We disable existing loggers to disable the "setup" logger used in the
    # parent process. This is needed on POSIX because the logger will
    # be there in the child following a fork().
    config_worker = {
        'version': 1,
        'disable_existing_loggers': True,
        'handlers': {
            'queue': {
                'class': 'logging.handlers.QueueHandler',
                'queue': q
            }
        },
        'root': {
            'handlers': ['queue'],
            'level': 'DEBUG'
        }
    }
    # The listener process configuration shows that the full flexibility of
    # logging configuration is available to dispatch events to handlers however
    # you want.
    # We disable existing loggers to disable the "setup" logger used in the
    # parent process. This is needed on POSIX because the logger will
    # be there in the child following a fork().
    config_listener = {
        'version': 1,
        'disable_existing_loggers': True,
        'formatters': {
            'detailed': {
                'class': 'logging.Formatter',
                'format': '%(asctime)s %(name)-15s %(levelname)-8s %(processName)-
↪10s %(message)s'
            },
            'simple': {
                'class': 'logging.Formatter',
                'format': '%(name)-15s %(levelname)-8s %(processName)-10s
↪%(message)s'
    }

```

```

    }
},
'handlers': {
    'console': {
        'class': 'logging.StreamHandler',
        'formatter': 'simple',
        'level': 'INFO'
    },
    'file': {
        'class': 'logging.FileHandler',
        'filename': 'mplog.log',
        'mode': 'w',
        'formatter': 'detailed'
    },
    'foofile': {
        'class': 'logging.FileHandler',
        'filename': 'mplog-foo.log',
        'mode': 'w',
        'formatter': 'detailed'
    },
    'errors': {
        'class': 'logging.FileHandler',
        'filename': 'mplog-errors.log',
        'mode': 'w',
        'formatter': 'detailed',
        'level': 'ERROR'
    }
},
'loggers': {
    'foo': {
        'handlers': ['foofile']
    }
},
'root': {
    'handlers': ['console', 'file', 'errors'],
    'level': 'DEBUG'
}
}

# Log some initial events, just to show that logging in the parent works
# normally.
logging.config.dictConfig(config_initial)
logger = logging.getLogger('setup')
logger.info('About to create workers ...')
workers = []
for i in range(5):
    wp = Process(target=worker_process, name='worker %d' % (i + 1),
                args=(config_worker,))
    workers.append(wp)
    wp.start()
    logger.info('Started worker: %s', wp.name)
logger.info('About to create listener ...')
stop_event = Event()
lp = Process(target=listener_process, name='listener',
            args=(q, stop_event, config_listener))
lp.start()
logger.info('Started listener')

```

```
# We now hang around for the workers to finish their work.
for wp in workers:
    wp.join()
# Workers all done, listening can now stop.
# Logging in the parent still works normally.
logger.info('Telling listener to stop ...')
stop_event.set()
lp.join()
logger.info('All done.')
```

```
if __name__ == '__main__':
    main()
```

21 SysLogHandler로 전송된 메시지에 BOM 삽입하기

RFC 5424 는 유니코드 메시지가 다음 구조를 갖는 바이트들로 syslog 데몬에 전송되어야 함을 요구합니다: 선택적인 순수 ASCII 구성 요소, 그 뒤를 이어 UTF-8 바이트 순서 포식 (BOM), 그 뒤를 이어 UTF-8으로 인코딩된 유니코드. (이 규격의 관련 절을 참조하십시오.)

파이썬 3.1에서, BOM을 메시지에 삽입하는 코드가 SysLogHandler 에 추가되었지만, 유감스럽게도, BOM 이 메시지의 시작 부분에 나타나서 그 앞에 순수 ASCII 구성 요소를 허락하지 않도록 잘못 구현되었습니다.

이 동작이 잘못됨에 따라, 잘못된 BOM 삽입 코드가 파이썬 3.2.4 이상에서 제거되었습니다. 그러나, 올바른 코드로 대체되지는 않았고, BOM을 포함하고, 그 앞에 순수 ASCII 시퀀스, 그 뒤에 UTF-8으로 인코딩된 임의의 유니코드로 구성된 **RFC 5424**-호환 메시지를 생성하려는 경우 다음과 같이 해야 합니다:

1. Formatter 인스턴스를 SysLogHandler 인스턴스에 다음과 같은 포맷 문자열과 함께 첨부하십시오:

```
'ASCII section\ufeffUnicode section'
```

유니코드 코드 포인트 U+FEFF는, UTF-8을 사용하여 인코딩될 때, UTF-8 BOM으로 인코딩됩니다 - 바이트열 b'\xef\xbb\xbf'.

2. ASCII section을 원하는 자리 표시기로 바꾸십시오. 그러나 치환 후 나타나는 데이터가 항상 ASCII 임미 보장되어야 합니다 (그렇게 되면, UTF-8 인코딩 이후에는 변경되지 않은 채로 유지됩니다).
3. Unicode section을 원하는 자리 표시기로 바꾸십시오; 치환 후 나타나는 데이터에 ASCII 범위를 벗어나는 문자가 포함되어 있어도 괜찮습니다 - UTF-8을 사용하여 인코딩됩니다.

포맷된 된 메시지는 SysLogHandler 에 의해 UTF-8 인코딩을 사용하여 인코딩*됩니다*. 위의 규칙을 따르는 경우, **RFC 5424**-호환 메시지를 생성할 수 있어야 합니다. 그렇지 않으면, logging이 불평하지 않을 수도 있지만, 메시지가 RFC 5424와 호환되지 않고 syslog 데몬이 불평 할 수 있습니다.

22 구조적 로깅 구현

대부분의 로깅 메시지는 사람이 읽을 수 있도록 만들어졌기 때문에 쉽게 기계에서 파싱 할 수 없지만, 프로그램에서 (복잡한 정규식을 사용하지 않고도) 구문 분석할 수 있는 구조화된 포맷으로 메시지를 출력하려는 상황이 있을 수 있습니다. 이것은 logging 패키지를 사용하여 쉽게 달성 할 수 있습니다. 이것이 달성될 수 있는 여러 가지 방법이 있지만, 다음은 JSON을 사용하여 기계가 파싱할 수 있는 방식으로 이벤트를 직렬화하는 간단한 접근법입니다:

```
import json
import logging

class StructuredMessage:
    def __init__(self, message, /, **kwargs):
        self.message = message
```

```

self.kwargs = kwargs

def __str__(self):
    return '%s >>> %s' % (self.message, json.dumps(self.kwargs))

_ = StructuredMessage # optional, to improve readability

logging.basicConfig(level=logging.INFO, format='%(message)s')
logging.info(_('message 1', foo='bar', bar='baz', num=123, fnum=123.456))

```

위의 스크립트가 실행되면 다음과 같이 인쇄됩니다:

```
message 1 >>> {"fnum": 123.456, "num": 123, "bar": "baz", "foo": "bar"}
```

항목의 순서는 사용된 파이썬 버전에 따라 다를 수 있습니다.

좀 더 특별한 처리가 필요한 경우, 다음 예제와 같이 사용자 정의 JSON 인코더를 사용할 수 있습니다:

```

import json
import logging

class Encoder(json.JSONEncoder):
    def default(self, o):
        if isinstance(o, set):
            return tuple(o)
        elif isinstance(o, str):
            return o.encode('unicode_escape').decode('ascii')
        return super().default(o)

class StructuredMessage:
    def __init__(self, message, /, **kwargs):
        self.message = message
        self.kwargs = kwargs

    def __str__(self):
        s = Encoder().encode(self.kwargs)
        return '%s >>> %s' % (self.message, s)

_ = StructuredMessage # optional, to improve readability

def main():
    logging.basicConfig(level=logging.INFO, format='%(message)s')
    logging.info(_('message 1', set_value={1, 2, 3}, snowman='\u2603'))

if __name__ == '__main__':
    main()

```

위의 스크립트를 실행하면 다음과 같이 인쇄합니다:

```
message 1 >>> {"snowman": "\u2603", "set_value": [1, 2, 3]}
```

항목의 순서는 사용된 파이썬 버전에 따라 다를 수 있습니다.

23 dictConfig() 로 처리기를 사용자 정의하기

특정 상황에서 로깅 처리기를 사용자 정의하고 싶을 때가 있고, dictConfig() 를 사용하고 있다면 서버 클래스를 만들지 않고도 이 작업을 수행 할 수 있습니다. 예를 들어, 로그 파일의 소유권을 설정하고 싶다고 합시다. POSIX에서, shutil.chown() 을 사용하면 쉽게 할 수 있지만, 표준 라이브러리의 파일 처리기는 내장된 지원을 제공하지 않습니다. 다음과 같은 일반 함수를 사용하여 처리기 생성을 사용자 정의 할 수 있습니다:

```
def owned_file_handler(filename, mode='a', encoding=None, owner=None):
    if owner:
        if not os.path.exists(filename):
            open(filename, 'a').close()
            shutil.chown(filename, *owner)
        return logging.FileHandler(filename, mode, encoding)
```

그런 다음, dictConfig() 에 전달되는 로깅 구성에서, 이 함수를 호출하여 로깅 처리기를 생성하도록 지정할 수 있습니다:

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'default': {
            'format': '%(asctime)s %(levelname)s %(name)s %(message)s'
        },
    },
    'handlers': {
        'file':{
            # The values below are popped from this dictionary and
            # used to create the handler, set the handler's level and
            # its formatter.
            '(): owned_file_handler,
            'level': 'DEBUG',
            'formatter': 'default',
            # The values below are passed to the handler creator callable
            # as keyword arguments.
            'owner': ['pulse', 'pulse'],
            'filename': 'chowntest.log',
            'mode': 'w',
            'encoding': 'utf-8',
        },
    },
    'root': {
        'handlers': ['file'],
        'level': 'DEBUG',
    },
}
```

이 예제에서는 단지 예를 들기 위해 pulse 라는 사용자와 그룹을 사용하여 소유권을 설정합니다. 작동하는 스크립트 chowntest.py 로 정리하면:

```
import logging, logging.config, os, shutil

def owned_file_handler(filename, mode='a', encoding=None, owner=None):
    if owner:
        if not os.path.exists(filename):
            open(filename, 'a').close()
            shutil.chown(filename, *owner)
```

(다음 페이지에 계속)

```

return logging.FileHandler(filename, mode, encoding)

LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'default': {
            'format': '%(asctime)s %(levelname)s %(name)s %(message)s'
        },
    },
    'handlers': {
        'file':{
            # The values below are popped from this dictionary and
            # used to create the handler, set the handler's level and
            # its formatter.
            '(): owned_file_handler,
            'level': 'DEBUG',
            'formatter': 'default',
            # The values below are passed to the handler creator callable
            # as keyword arguments.
            'owner': ['pulse', 'pulse'],
            'filename': 'chowntest.log',
            'mode': 'w',
            'encoding': 'utf-8',
        },
    },
    'root': {
        'handlers': ['file'],
        'level': 'DEBUG',
    },
}

logging.config.dictConfig(LOGGING)
logger = logging.getLogger('mylogger')
logger.debug('A debug message')

```

이것을 실행하기 위해서는, 아마도 root 로 실행해야 할 것입니다:

```

$ sudo python3.3 chowntest.py
$ cat chowntest.log
2013-11-05 09:34:51,128 DEBUG mylogger A debug message
$ ls -l chowntest.log
-rw-r--r-- 1 pulse pulse 55 2013-11-05 09:34 chowntest.log

```

이 예제는 `shutil.chown()` 이 등장한 파이썬 3.3을 사용합니다. 이 접근법은 `dictConfig()` 를 지원하는 모든 파이썬 버전에서 작동합니다- 파이썬 2.7, 3.2 이상. 3.3 이전 버전의 경우, (예를 들어) `os.chown()` 을 사용하여 실제 소유권 변경을 구현해야 합니다.

실제로는, 처리기 생성 함수가 프로젝트 어딘가에 있는 유틸리티 모듈에 있을 수 있습니다. 구성에 있는 다음과 같은 줄 대신:

```
'(): owned_file_handler,
```

예를 들면 이렇게 쓸 수 있습니다:

```
'(): 'ext://project.util.owned_file_handler',
```

여기서 `project.util` 은 함수가 있는 패키지의 실제 이름으로 바꿀 수 있습니다. 위의 작업 스크립트에서

'ext://__main__.owned_file_handler' 를 사용해도 됩니다. 여기서, 실제 콜러블은 ext:// 스펙으로부터 dictConfig() 에 의해 결정됩니다.

이 예제는 희망하건대 다른 형태의 파일 변경을 - 예를 들어 특정 POSIX 권한 비트 설정 - 같은 방법으로 (os.chmod() 를 사용해서) 구현하는 방법도 알려줍니다.

물론 이 접근법은 FileHandler 이외의 처리기 유형으로도 확장될 수 있습니다 - 예를 들어, 회전 파일 처리기 중 하나 또는 다른 유형의 처리기 모두.

24 응용 프로그램 전체에서 특정 포맷 스타일 사용하기

파이썬 3.2에서, Formatter 는 style 키워드 매개변수를 얻었는데, 이전 버전과의 호환성을 위해 % 를 기본값으로 사용하면서 { 또는 \$ 를 지정하면 str.format() 과 string.Template 에 의해 지원되는 포맷팅 접근법을 사용할 수 있도록 합니다. 이것은 로그 되는 최종 출력으로 로깅 메시지를 포맷팅하는 것과 관계된 것이고, 개별 로깅 메시지가 만들어지는 방법과는 무관함에 주의하십시오.

Logging calls (debug(), info() etc.) only take positional parameters for the actual logging message itself, with keyword parameters used only for determining options for how to handle the logging call (e.g. the exc_info keyword parameter to indicate that traceback information should be logged, or the extra keyword parameter to indicate additional contextual information to be added to the log). So you cannot directly make logging calls using str.format() or string.Template syntax, because internally the logging package uses %-formatting to merge the format string and the variable arguments. There would be no changing this while preserving backward compatibility, since all logging calls which are out there in existing code will be using %-format strings.

포맷 스타일을 특정 로거와 연관시키는 제안이 있었지만, 이전 버전과의 호환성 문제가 있는데, 기존 코드가 그 로거 이름으로 %-포맷팅을 사용할 수 있기 때문입니다.

제삼자 라이브러리와 여러분의 코드 간에 상호 운용이 가능하도록 로깅 하려면, 개별 로깅 호출 수준에서 포맷팅을 결정해야 합니다. 이렇게 할 때 대체 포맷팅 스타일을 수용 할 수 있는 몇 가지 길이 열립니다.

24.1 LogRecord 팩토리 사용

파이썬 3.2에서, 위에서 언급 한 Formatter 변경 사항과 함께, logging 패키지는 setLogRecordFactory() 함수를 사용하여 사용자가 자신의 LogRecord 서브 클래스를 설정할 수 있는 기능을 얻었습니다. 이것을 사용하면, 원하는 일을 하도록 getMessage() 메서드를 재정의하는 여러분 자신의 LogRecord 서브 클래스를 설정할 수 있습니다. 이 메서드의 베이스 클래스 구현이 msg % args 포맷팅이 일어나는 곳이며, 여러분이 대체 포맷팅으로 치환할 수 있는 곳입니다; 그러나, 모든 포맷팅 스타일을 지원하면서 다른 코드와의 상호 운용성을 보장하기 위해 %-포맷팅을 기본값으로 사용하도록 주의해야 합니다. 또한, 베이스 구현과 마찬가지로 str(self.msg) 를 호출하도록 주의해야 합니다.

자세한 정보는 setLogRecordFactory() 와 LogRecord 에 대한 레퍼런스 설명서를 참조하십시오.

24.2 사용자 정의 메시지 객체 사용

{- 및 \$-포맷팅을 사용하여 개별 로그 메시지를 작성할 수 있는 또 다른, 아마도 더 간단한 방법이 있습니다. (arbitrary-object-messages에서) 로깅 할 때 임의의 객체를 메시지 포맷 문자열로 사용할 수 있고, logging 패키지는 그 객체에 대해 str() 을 호출하여 실제 형식 문자열을 얻는다고 했던 것을 기억하실 수 있을 겁니다. 다음 두 클래스를 생각해봅시다:

```
class BraceMessage:
    def __init__(self, fmt, /, *args, **kwargs):
        self.fmt = fmt
        self.args = args
        self.kwargs = kwargs

    def __str__(self):
        return self.fmt.format(*self.args, **self.kwargs)

class DollarMessage:
    def __init__(self, fmt, /, **kwargs):
```

(다음 페이지에 계속)

```

self.fmt = fmt
self.kwargs = kwargs

def __str__(self):
    from string import Template
    return Template(self.fmt).substitute(**self.kwargs)

```

이 중 하나를 포맷 문자열 대신 사용하면, {}- 또는 \$-포매팅을 사용하여 포맷된 로그 출력의 “%(message)s”, “{message}” 또는 “\$message” 자리에 나타나는 실제 “message” 부분을 만들 수 있습니다. 어떤 것을 로그 하고 싶을 때마다 클래스 이름을 사용하는 것이 다소 꼴사납다면, 메시지에 M 이나 _ 과 같은 별칭을 사용해서 더 쓸만하게 만들 수 있습니다 (또는 지역화에 _ 를 사용하고 있다면, 아마도 __).

이 접근법의 예가 아래에 나와 있습니다. 먼저, str.format() 를 사용하는 포매팅입니다:

```

>>> __ = BraceMessage
>>> print(__('Message with {0} {1}', 2, 'placeholders'))
Message with 2 placeholders
>>> class Point: pass
...
>>> p = Point()
>>> p.x = 0.5
>>> p.y = 0.5
>>> print(__('Message with coordinates: ({point.x:.2f}, {point.y:.2f})', point=p))
Message with coordinates: (0.50, 0.50)

```

두 번째로, string.Template 를 사용하는 포매팅입니다:

```

>>> __ = DollarMessage
>>> print(__('Message with $num $what', num=2, what='placeholders'))
Message with 2 placeholders
>>>

```

One thing to note is that you pay no significant performance penalty with this approach: the actual formatting happens not when you make the logging call, but when (and if) the logged message is actually about to be output to a log by a handler. So the only slightly unusual thing which might trip you up is that the parentheses go around the format string and the arguments, not just the format string. That’s because the __ notation is just syntax sugar for a constructor call to one of the XXXMessage classes shown above.

25 dictConfig() 로 필터 구성하기

dictConfig() 를 사용하여 필터를 구성할 수 있습니다. 하지만 처음에는 어떻게 해야 할지 명확하지 않을 수 있습니다 (그래서 이 조리법을 제공합니다). Filter 가 표준 라이브러리에 포함된 유일한 필터 클래스 이고, 많은 요구 사항을 충족시키지는 않을 것이기 때문에 (오직 베이스 클래스로 제공됩니다), 일반적으로 filter() 메서드를 재정의하는 여러분 자신의 Filter 서브 클래스를 정의할 필요가 있습니다. 이렇게 하려면, 필터를 생성하는 데 사용될 콜러블을 필터의 구성 디렉터리에 () 키로 지정하십시오 (클래스가 가장 분명하지만 Filter 인스턴스를 반환하는 콜러블은 모두 가능합니다). 다음은 완전한 예입니다:

```

import logging
import logging.config
import sys

class MyFilter(logging.Filter):
    def __init__(self, param=None):
        self.param = param

    def filter(self, record):

```

```

    if self.param is None:
        allow = True
    else:
        allow = self.param not in record.msg
    if allow:
        record.msg = 'changed: ' + record.msg
    return allow

LOGGING = {
    'version': 1,
    'filters': {
        'myfilter': {
            '()': MyFilter,
            'param': 'noshow',
        }
    },
    'handlers': {
        'console': {
            'class': 'logging.StreamHandler',
            'filters': ['myfilter']
        }
    },
    'root': {
        'level': 'DEBUG',
        'handlers': ['console']
    },
}

if __name__ == '__main__':
    logging.config.dictConfig(LOGGING)
    logging.debug('hello')
    logging.debug('hello - noshow')

```

이 예제는 인스턴스를 만드는 콜러블로 키워드 매개 변수 형식으로 구성 데이터를 전달하는 방법을 보여줍니다. 실행하면, 위의 스크립트는 다음을 인쇄합니다:

```
changed: hello
```

필터가 구성된 대로 작동하고 있음을 보여줍니다.

주목해야 할 몇 가지 추가 사항:

- 구성에서 직접 참조할 수 없는 경우 (예를 들어, 다른 모듈에 있고 구성 디렉터리가 있는 곳에서 직접 임포트 할 수 없는 경우), `logging-config-dict-externalobj` 에 설명된 대로 `ext://...` 형식을 사용할 수 있습니다. 예를 들어, 위의 예에서 `MyFilter` 대신 `'ext://__main__.MyFilter'` 를 사용할 수 있습니다.
- 필터뿐만 아니라, 이 기술을 사용자 정의 처리기 및 포매터를 구성하는데 사용할 수도 있습니다. `logging` 이 구성에서 사용자 정의 객체를 어떻게 지원하는지에 대한 더 많은 정보는 `logging-config-dict-userdef` 를 보시고, 위의 다른 요리책 조리법 `dictConfig()` 로 처리기를 사용자 정의하기도 보십시오.

26 사용자 정의된 예외 포매팅

예외 포매팅을 사용자 정의하고 싶을 때가 있습니다 - 논쟁의 여지는 있지만, 예외 정보가 포함된 경우에도 이벤트 당 정확히 한 줄이 기록되기 원한다고 합시다. 다음 예제처럼, 사용자 정의 포매터 클래스를 사용할 수 있습니다:

```

import logging

class OneLineExceptionFormatter(logging.Formatter):
    def formatException(self, exc_info):
        """
        Format an exception so that it prints on a single line.
        """
        result = super().formatException(exc_info)
        return repr(result) # or format into one line however you want to

    def format(self, record):
        s = super().format(record)
        if record.exc_text:
            s = s.replace('\n', '|') + '|'
        return s

def configure_logging():
    fh = logging.FileHandler('output.txt', 'w')
    f = OneLineExceptionFormatter('%(asctime)s|%(levelname)s|%(message)s|',
                                  '%d/%m/%Y %H:%M:%S')

    fh.setFormatter(f)
    root = logging.getLogger()
    root.setLevel(logging.DEBUG)
    root.addHandler(fh)

def main():
    configure_logging()
    logging.info('Sample message')
    try:
        x = 1 / 0
    except ZeroDivisionError as e:
        logging.exception('ZeroDivisionError: %s', e)

if __name__ == '__main__':
    main()

```

실행하면, 정확하게 두 줄의 파일이 생성됩니다:

```

28/01/2015 07:21:23|INFO|Sample message|
28/01/2015 07:21:23|ERROR|ZeroDivisionError: division by zero|'Traceback (most
↵recent call last):\n  File "logtest7.py", line 30, in main\n    x = 1 / 0\
↵\nZeroDivisionError: division by zero'|

```

위의 처리는 단순하지만, 예외 정보를 원하는 대로 포맷하는 방법을 알려줍니다. traceback 모듈은 더욱 전문화된 요구에 도움이 될 수 있습니다.

27 로깅 메시지 말하기

로깅 메시지를 보여주는 대신 들려주는 것이 바람직한 상황이 있을 수 있습니다. 여러분의 시스템에 텍스트-음성 변환 (TTS) 기능이 있다면 쉽습니다, 파이썬 바인딩이 없어도 됩니다. 대부분의 TTS 시스템에는 실행할 수 있는 명령행 프로그램이 있으며, 이것을 subprocess 를 사용하여 처리기에서 호출 할 수 있습니다. 여기서 TTS 명령행 프로그램이 사용자와 상호 작용하거나, 완료하는데 오랜 시간이 걸릴 것으로 기대 되지 않으며, 로그 되는 메시지의 빈도가 메시지로 사용자를 압도할 정도로 높지 않으며, 메시지는 동시에 처리되지 않고 한 번에 하나씩 읽어도 된다고 가정합니다. 아래의 예제 구현은 다음 메시지가 처리되기 전에 하나의 메시지를 다 읽을 때까지 대기하고, 이 때문에 다른 처리기가 대기 상태로 유지될 수 있습니다. 다음은 espeak TTS 패키지가 사용 가능하다고 가정하는 접근법을 보여주는 간단한 예입니다:

```

import logging
import subprocess
import sys

class TTSHandler(logging.Handler):
    def emit(self, record):
        msg = self.format(record)
        # Speak slowly in a female English voice
        cmd = ['espeak', '-s150', '-ven+f3', msg]
        p = subprocess.Popen(cmd, stdout=subprocess.PIPE,
                              stderr=subprocess.STDOUT)

        # wait for the program to finish
        p.communicate()

def configure_logging():
    h = TTSHandler()
    root = logging.getLogger()
    root.addHandler(h)
    # the default formatter just returns the message
    root.setLevel(logging.DEBUG)

def main():
    logging.info('Hello')
    logging.debug('Goodbye')

if __name__ == '__main__':
    configure_logging()
    sys.exit(main())

```

실행하면, 이 스크립트는 여성 음성으로 “Hello”와 “Goodbye”를 차례대로 말합니다.

물론 위의 접근법은 다른 TTS 시스템과 명령행에서 실행되는 외부 프로그램을 통해 메시지를 처리 할 수 있는 전혀 다른 시스템에도 적용될 수 있습니다.

28 로깅 메시지를 버퍼링하고 조건부 출력하기

임시 영역에 메시지를 기록하고 특정 조건이 발생할 때만 메시지를 출력하려는 상황이 있을 수 있습니다. 예를 들어, 함수에서 디버그 이벤트를 로깅하기를 원할 수 있습니다. 함수가 예러 없이 완료되면 수집된 디버그 정보로 로그를 어지럽히고 싶지 않지만, 예러가 있으면 예러뿐만 아니라 모든 디버그 정보를 출력하고 싶습니다.

다음은 로깅이 이러한 방식으로 작동하기 원하는 함수에 데코레이터를 사용하여 이를 수행할 방법을 보여주는 예제입니다. `logging.handlers.MemoryHandler`를 사용하는데, 어떤 상황이 발생할 때까지 로그된 이벤트를 버퍼링할 수 있도록 하고, 때가 되면 버퍼링된 이벤트들이 flush 됩니다- 처리를 위해 다른 처리기(target 처리기)로 전달됩니다. 기본적으로, `MemoryHandler`는 버퍼가 다 차거나 수준이 지정된 임계값보다 크거나 같은 이벤트가 발생하면 플러시 됩니다. 사용자 정의 플러시 동작을 원할 경우, 이 조리법을 `MemoryHandler`의 더 특수한 서브 클래스와 함께 사용할 수 있습니다.

예제 스크립트에는 간단한 함수 `foo`가 있는데, 모든 로그 수준을 순회하면서, 어떤 수준으로 로그 할지를 `sys.stderr`에 쓴 다음, 그 수준으로 실제 메시지를 로깅 합니다. 매개 변수를 `foo`에 전달할 수 있는데, 참이면 `ERROR` 및 `CRITICAL` 수준으로 로그 합니다- 그렇지 않으면 `DEBUG`, `INFO` 및 `WARNING` 수준에서만 로그 합니다.

이 스크립트는 필요한 조건부 로깅을 수행할 데코레이터로 `foo`를 데코레이트 하기만 합니다. 데코레이터는 로거를 매개 변수로 받고 데코레이트 된 함수가 호출되는 동안 메모리 처리기를 연결합니다. 데코레이터는 target 처리기, 플러싱이 발생해야 하는 수준 및 버퍼 용량(버퍼 된 레코드의 수)을 추가로 매개 변수로 받을 수 있습니다. 이것들은 각각 `sys.stderr`로 쓰는 `StreamHandler`, `logging.ERROR`, 100을 기본값으로 합니다.

스크립트는 다음과 같습니다:

```
import logging
from logging.handlers import MemoryHandler
import sys

logger = logging.getLogger(__name__)
logger.addHandler(logging.NullHandler())

def log_if_errors(logger, target_handler=None, flush_level=None, capacity=None):
    if target_handler is None:
        target_handler = logging.StreamHandler()
    if flush_level is None:
        flush_level = logging.ERROR
    if capacity is None:
        capacity = 100
    handler = MemoryHandler(capacity, flushLevel=flush_level, target=target_
↪handler)

    def decorator(fn):
        def wrapper(*args, **kwargs):
            logger.addHandler(handler)
            try:
                return fn(*args, **kwargs)
            except Exception:
                logger.exception('call failed')
                raise
            finally:
                super(MemoryHandler, handler).flush()
                logger.removeHandler(handler)
        return wrapper

    return decorator

def write_line(s):
    sys.stderr.write('%s\n' % s)

def foo(fail=False):
    write_line('about to log at DEBUG ...')
    logger.debug('Actually logged at DEBUG')
    write_line('about to log at INFO ...')
    logger.info('Actually logged at INFO')
    write_line('about to log at WARNING ...')
    logger.warning('Actually logged at WARNING')
    if fail:
        write_line('about to log at ERROR ...')
        logger.error('Actually logged at ERROR')
        write_line('about to log at CRITICAL ...')
        logger.critical('Actually logged at CRITICAL')
    return fail

decorated_foo = log_if_errors(logger)(foo)

if __name__ == '__main__':
    logger.setLevel(logging.DEBUG)
    write_line('Calling undecorated foo with False')
    assert not foo(False)
    write_line('Calling undecorated foo with True')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
assert foo(True)
write_line('Calling decorated foo with False')
assert not decorated_foo(False)
write_line('Calling decorated foo with True')
assert decorated_foo(True)
```

이 스크립트를 실행하면 다음과 같은 출력이 나타납니다.:

```
Calling undecorated foo with False
about to log at DEBUG ...
about to log at INFO ...
about to log at WARNING ...
Calling undecorated foo with True
about to log at DEBUG ...
about to log at INFO ...
about to log at WARNING ...
about to log at ERROR ...
about to log at CRITICAL ...
Calling decorated foo with False
about to log at DEBUG ...
about to log at INFO ...
about to log at WARNING ...
Calling decorated foo with True
about to log at DEBUG ...
about to log at INFO ...
about to log at WARNING ...
about to log at ERROR ...
Actually logged at DEBUG
Actually logged at INFO
Actually logged at WARNING
Actually logged at ERROR
about to log at CRITICAL ...
Actually logged at CRITICAL
```

보시다시피, 실제 로깅 출력은 심각도가 **ERROR** 이상인 이벤트가 기록될 때만 발생하지만, 이 경우 심각도가 낮은 이전 이벤트도 기록됩니다.

물론 전통적인 데코레이션 수단을 쓸 수 있습니다.:

```
@log_if_errors(logger)
def foo(fail=False):
    ...
```

29 Sending logging messages to email, with buffering

To illustrate how you can send log messages via email, so that a set number of messages are sent per email, you can subclass `BufferingHandler`. In the following example, which you can adapt to suit your specific needs, a simple test harness is provided which allows you to run the script with command line arguments specifying what you typically need to send things via SMTP. (Run the downloaded script with the `-h` argument to see the required and optional arguments.)

```
import logging
import logging.handlers
import smtplib

class BufferingSMTPHandler(logging.handlers.BufferingHandler):
```

(다음 페이지에 계속)

```

def __init__(self, mailhost, port, username, password, fromaddr, toaddrs,
             subject, capacity):
    logging.handlers.BufferingHandler.__init__(self, capacity)
    self.mailhost = mailhost
    self.mailport = port
    self.username = username
    self.password = password
    self.fromaddr = fromaddr
    if isinstance(toaddrs, str):
        toaddrs = [toaddrs]
    self.toaddrs = toaddrs
    self.subject = subject
    self.setFormatter(logging.Formatter("%(asctime)s %(levelname)-5s
↪ %(message)s"))

def flush(self):
    if len(self.buffer) > 0:
        try:
            smtp = smtplib.SMTP(self.mailhost, self.mailport)
            smtp.starttls()
            smtp.login(self.username, self.password)
            msg = "From: %s\r\nTo: %s\r\nSubject: %s\r\n\r\n" % (self.fromaddr,
↪ ', '.join(self.toaddrs), self.subject)
            for record in self.buffer:
                s = self.format(record)
                msg = msg + s + "\r\n"
            smtp.sendmail(self.fromaddr, self.toaddrs, msg)
            smtp.quit()
        except Exception:
            if logging.raiseExceptions:
                raise
            self.buffer = []

if __name__ == '__main__':
    import argparse

    ap = argparse.ArgumentParser()
    aa = ap.add_argument
    aa('host', metavar='HOST', help='SMTP server')
    aa('--port', '-p', type=int, default=587, help='SMTP port')
    aa('user', metavar='USER', help='SMTP username')
    aa('password', metavar='PASSWORD', help='SMTP password')
    aa('to', metavar='TO', help='Addressee for emails')
    aa('sender', metavar='SENDER', help='Sender email address')
    aa('--subject', '-s',
        default='Test Logging email from Python logging module (buffering)',
        help='Subject of email')
    options = ap.parse_args()
    logger = logging.getLogger()
    logger.setLevel(logging.DEBUG)
    h = BufferingSMTPHandler(options.host, options.port, options.user,
                            options.password, options.sender,
                            options.to, options.subject, 10)

    logger.addHandler(h)
    for i in range(102):
        logger.info("Info index = %d", i)

```

```
h.flush()
h.close()
```

If you run this script and your SMTP server is correctly set up, you should find that it sends eleven emails to the addressee you specify. The first ten emails will each have ten log messages, and the eleventh will have two messages. That makes up 102 messages as specified in the script.

30 구성을 통해 UTC(GMT)로 시간을 포맷하기

Sometimes you want to format times using UTC, which can be done using a class such as `UTCFormatter`, shown below:

```
import logging
import time

class UTCFormatter(logging.Formatter):
    converter = time.gmtime
```

이제 `Formatter` 대신 코드에서 `UTCFormatter` 를 사용할 수 있습니다. 구성을 통해 이를 수행하려면, 다음에 나오는 완전한 예제에 의해 설명된 접근법으로 `dictConfig()` API를 사용할 수 있습니다:

```
import logging
import logging.config
import time

class UTCFormatter(logging.Formatter):
    converter = time.gmtime

LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'utc': {
            '()': UTCFormatter,
            'format': '%(asctime)s %(message)s',
        },
        'local': {
            'format': '%(asctime)s %(message)s',
        }
    },
    'handlers': {
        'console1': {
            'class': 'logging.StreamHandler',
            'formatter': 'utc',
        },
        'console2': {
            'class': 'logging.StreamHandler',
            'formatter': 'local',
        },
    },
    'root': {
        'handlers': ['console1', 'console2'],
    }
}

if __name__ == '__main__':
```

```
logging.config.dictConfig(LOGGING)
logging.warning('The local time is %s', time.asctime())
```

이 스크립트를 실행하면, 다음과 같은 내용을 인쇄합니다:

```
2015-10-17 12:53:29,501 The local time is Sat Oct 17 13:53:29 2015
2015-10-17 13:53:29,501 The local time is Sat Oct 17 13:53:29 2015
```

시간이 한 처리기에서는 UTC로, 다른 처리기에서는 지역 시간으로 포맷되는 것을 보여줍니다.

31 선택적 로깅을 위해 컨텍스트 관리자 사용하기

로깅 구성을 일시적으로 변경하고 무언가를 한 후에 되돌리는 것이 유용할 때가 있습니다. 이를 위해, 컨텍스트 관리자는 로깅 컨텍스트를 저장하고 복원하는 가장 분명한 방법입니다. 다음은 그러한 컨텍스트 관리자의 간단한 예입니다. 컨텍스트 관리자의 범위 안에서 선택적으로 로깅 수준을 변경하고 로깅 처리기를 추가 할 수 있습니다:

```
import logging
import sys

class LoggingContext:
    def __init__(self, logger, level=None, handler=None, close=True):
        self.logger = logger
        self.level = level
        self.handler = handler
        self.close = close

    def __enter__(self):
        if self.level is not None:
            self.old_level = self.logger.level
            self.logger.setLevel(self.level)
        if self.handler:
            self.logger.addHandler(self.handler)

    def __exit__(self, et, ev, tb):
        if self.level is not None:
            self.logger.setLevel(self.old_level)
        if self.handler:
            self.logger.removeHandler(self.handler)
        if self.handler and self.close:
            self.handler.close()
        # implicit return of None => don't swallow exceptions
```

수준 값을 지정하면, 로거의 수준은 컨텍스트 관리자가 적용되는 with 블록의 범위 안에서 해당 값으로 설정됩니다. 처리기를 지정하면, 블록 진입 시 로거에 추가되고 블록에서 빠져나갈 때 제거됩니다. 블록을 빠져나갈 때 처리기를 닫도록 관리자에게 요청할 수도 있습니다 - 더는 처리기가 필요하지 않으면 이렇게 할 수 있습니다.

작동 원리를 보여주기 위해, 다음 코드 블록을 위에 추가 할 수 있습니다:

```
if __name__ == '__main__':
    logger = logging.getLogger('foo')
    logger.addHandler(logging.StreamHandler())
    logger.setLevel(logging.INFO)
    logger.info('1. This should appear just once on stderr.')
    logger.debug('2. This should not appear.')
```

```

with LoggingContext(logger, level=logging.DEBUG):
    logger.debug('3. This should appear once on stderr.')
logger.debug('4. This should not appear.')
h = logging.StreamHandler(sys.stdout)
with LoggingContext(logger, level=logging.DEBUG, handler=h, close=True):
    logger.debug('5. This should appear twice - once on stderr and once on
↳stdout.')
logger.info('6. This should appear just once on stderr.')
logger.debug('7. This should not appear.')

```

우리는 초기에 로거 수준을 INFO 로 설정합니다. 그래서 메시지 #1은 나타나고 메시지 #2는 나타나지 않습니다. 그다음에 with 블록에서 수준을 DEBUG 로 임시 변경하면, 메시지 #3이 나타납니다. 블록이 종료되면 로거 수준이 INFO로 복원되므로, 메시지 #4가 표시되지 않습니다. 그다음 with 블록에서 수준을 다시 DEBUG 로 다시 설정하지만, sys.stdout 으로 쓰는 처리기도 추가합니다. 따라서 메시지 #5는 콘솔에 두 번 표시됩니다 (stderr 를 통해 한 번, stdout 을 통해 한 번). with 문장이 완료된 후에 상태는 이전과 같으므로, (메시지 #1처럼) 메시지 #6이 나타나고, (메시지 #2처럼) 메시지 #7은 보이지 않습니다.

이렇게 만든 스크립트를 실행하면, 결과는 다음과 같습니다:

```

$ python logctx.py
1. This should appear just once on stderr.
3. This should appear once on stderr.
5. This should appear twice - once on stderr and once on stdout.
5. This should appear twice - once on stderr and once on stdout.
6. This should appear just once on stderr.

```

다시 실행하면서 stderr 를 /dev/null 로 리디렉트하면, 다음과 같이 stdout 으로 출력된 메시지만 나타납니다:

```

$ python logctx.py 2>/dev/null
5. This should appear twice - once on stderr and once on stdout.

```

다시 한번, 하지만 stdout 을 /dev/null 로 리디렉트하면, 이렇게 됩니다:

```

$ python logctx.py >/dev/null
1. This should appear just once on stderr.
3. This should appear once on stderr.
5. This should appear twice - once on stderr and once on stdout.
6. This should appear just once on stderr.

```

이 경우, stdout 에 인쇄된 메시지 #5는 예상대로 나타나지 않습니다.

물론 여기서 설명한 방법을 일반화 할 수 있습니다. 예를 들어 로깅 필터를 임시로 첨부 할 수 있습니다. 위의 코드는 파이썬 2와 파이썬 3에서 모두 작동합니다.

32 CLI 응용 프로그램 시작 템플릿

다음과 같은 것들을 하는 방법을 보여주는 예입니다:

- 명령 줄 인자에 기반한 로깅 수준 사용하기
- 별도의 파일에 있는 여러 부속 명령으로 분기하고, 모두 일관된 방식으로 같은 수준에서 로깅 하기
- 간단하고 최소의 구성을 사용하기

어떤 서비스를 중지, 시작 또는 다시 시작하는 작업을 위한 명령 줄 응용 프로그램이 있다고 가정합니다. 이것은 예시의 목적을 위해 start.py, stop.py 및 restart.py에 개별 명령이 구현되고, 응용 프로그램의 메인 스크립트는 app.py 파일이 되도록 구성할 수 있습니다. 명령 줄 인자를 통해 응용 프로그램의 상세도를 제어하려고 하고, logging.INFO를 기본값으로 한다고 더 가정해 봅시다. app.py를 작성할 수 있는 한 가지 방법은 다음과 같습니다:

```

import argparse
import importlib
import logging
import os
import sys

def main(args=None):
    scriptname = os.path.basename(__file__)
    parser = argparse.ArgumentParser(scriptname)
    levels = ('DEBUG', 'INFO', 'WARNING', 'ERROR', 'CRITICAL')
    parser.add_argument('--log-level', default='INFO', choices=levels)
    subparsers = parser.add_subparsers(dest='command',
                                       help='Available commands:')

    start_cmd = subparsers.add_parser('start', help='Start a service')
    start_cmd.add_argument('name', metavar='NAME',
                          help='Name of service to start')

    stop_cmd = subparsers.add_parser('stop',
                                     help='Stop one or more services')
    stop_cmd.add_argument('names', metavar='NAME', nargs='+',
                          help='Name of service to stop')

    restart_cmd = subparsers.add_parser('restart',
                                       help='Restart one or more services')
    restart_cmd.add_argument('names', metavar='NAME', nargs='+',
                             help='Name of service to restart')

    options = parser.parse_args()
    # the code to dispatch commands could all be in this file. For the purposes
    # of illustration only, we implement each command in a separate module.
    try:
        mod = importlib.import_module(options.command)
        cmd = getattr(mod, 'command')
    except (ImportError, AttributeError):
        print('Unable to find the code for command \'%s\'' % options.command)
        return 1
    # Could get fancy here and load configuration from file or dictionary
    logging.basicConfig(level=options.log_level,
                        format='%(levelname)s %(name)s %(message)s')
    cmd(options)

if __name__ == '__main__':
    sys.exit(main())

```

그리고 start, stop 및 restart 명령은 별도의 모듈로 구현할 수 있습니다, 가령 시작하려면:

```

# start.py
import logging

logger = logging.getLogger(__name__)

def command(options):
    logger.debug('About to start %s', options.name)
    # actually do the command processing here ...
    logger.info('Started the \'%s\' service.', options.name)

```

그리고 멈추려면:

```

# stop.py
import logging

```

(다음 페이지에 계속)

```

logger = logging.getLogger(__name__)

def command(options):
    n = len(options.names)
    if n == 1:
        plural = ''
        services = '\\'%s\\' % options.names[0]
    else:
        plural = 's'
        services = ', '.join('\\'%s\\' % name for name in options.names)
        i = services.rfind(', ')
        services = services[:i] + ' and ' + services[i + 2:]
    logger.debug('About to stop %s', services)
    # actually do the command processing here ...
    logger.info('Stopped the %s service%s.', services, plural)

```

비슷하게, 다시 시작하려면:

```

# restart.py
import logging

logger = logging.getLogger(__name__)

def command(options):
    n = len(options.names)
    if n == 1:
        plural = ''
        services = '\\'%s\\' % options.names[0]
    else:
        plural = 's'
        services = ', '.join('\\'%s\\' % name for name in options.names)
        i = services.rfind(', ')
        services = services[:i] + ' and ' + services[i + 2:]
    logger.debug('About to restart %s', services)
    # actually do the command processing here ...
    logger.info('Restarted the %s service%s.', services, plural)

```

이 응용 프로그램을 기본 로그 수준으로 실행하면, 이런 출력을 얻습니다:

```

$ python app.py start foo
INFO start Started the 'foo' service.

$ python app.py stop foo bar
INFO stop Stopped the 'foo' and 'bar' services.

$ python app.py restart foo bar baz
INFO restart Restarted the 'foo', 'bar' and 'baz' services.

```

첫 번째 단어는 로그 수준이고, 두 번째 단어는 이벤트가 로그된 장소의 모듈이나 패키지 이름입니다.

로그 수준을 변경하면, 로그로 전송되는 정보를 변경할 수 있습니다. 예를 들어, 우리가 더 많은 정보를 원한다면:

```

$ python app.py --log-level DEBUG start foo
DEBUG start About to start foo
INFO start Started the 'foo' service.

```

```
$ python app.py --log-level DEBUG stop foo bar
DEBUG stop About to stop 'foo' and 'bar'
INFO stop Stopped the 'foo' and 'bar' services.

$ python app.py --log-level DEBUG restart foo bar baz
DEBUG restart About to restart 'foo', 'bar' and 'baz'
INFO restart Restarted the 'foo', 'bar' and 'baz' services.
```

그리고 덜 원한다면:

```
$ python app.py --log-level WARNING start foo
$ python app.py --log-level WARNING stop foo bar
$ python app.py --log-level WARNING restart foo bar baz
```

이 경우, WARNING 수준 이상으로 아무것도 로그 하지 않았으므로, 명령은 콘솔에 아무것도 인쇄하지 않습니다.

33 로깅을 위한 Qt GUI

A question that comes up from time to time is about how to log to a GUI application. The Qt framework is a popular cross-platform UI framework with Python bindings using PySide2 or PyQt5 libraries.

다음 예는 Qt GUI에 로그 하는 방법을 보여줍니다. 이것은 콜러블을 취하는 간단한 QtHandler 클래스를 소개합니다. 이 클래스는 GUI 업데이트를 하는 메인 스레드의 슬롯이어야 합니다. 또한 UI 자체 (수동 로깅을 위한 버튼을 통해) 뿐만 아니라 백그라운드에서 작업하는 작업자 스레드에서 GUI에 로그 하는 방법을 보여주기 위해 작업자 스레드도 만듭니다 (여기서는, 단지 임의의 짧은 지연을 주고 임의의 수준으로 메시지를 로깅 합니다).

작업자 스레드는 threading 모듈 대신 Qt의 QThread 클래스를 사용하여 구현되는데, 다른 Qt 구성 요소와 더 잘 통합되는 QThread를 사용해야 하는 상황이 있기 때문입니다.

The code should work with recent releases of any of PySide6, PyQt6, PySide2 or PyQt5. You should be able to adapt the approach to earlier versions of Qt. Please refer to the comments in the code snippet for more detailed information.

```
import datetime
import logging
import random
import sys
import time

# Deal with minor differences between different Qt packages
try:
    from PySide6 import QtCore, QtGui, QtWidgets
    Signal = QtCore.Signal
    Slot = QtCore.Slot
except ImportError:
    try:
        from PyQt6 import QtCore, QtGui, QtWidgets
        Signal = QtCore.pyqtSignal
        Slot = QtCore.pyqtSlot
    except ImportError:
        try:
            from PySide2 import QtCore, QtGui, QtWidgets
            Signal = QtCore.Signal
            Slot = QtCore.Slot
```

(다음 페이지에 계속)

```

except ImportError:
    from PyQt5 import QtCore, QtGui, QtWidgets
    Signal = QtCore.pyqtSignal
    Slot = QtCore.pyqtSlot

logger = logging.getLogger(__name__)

#
# Signals need to be contained in a QObject or subclass in order to be correctly
# initialized.
#
class Signaller(QtCore.QObject):
    signal = Signal(str, logging.LogRecord)

#
# Output to a Qt GUI is only supposed to happen on the main thread. So, this
# handler is designed to take a slot function which is set up to run in the main
# thread. In this example, the function takes a string argument which is a
# formatted log message, and the log record which generated it. The formatted
# string is just a convenience - you could format a string for output any way
# you like in the slot function itself.
#
# You specify the slot function to do whatever GUI updates you want. The handler
# doesn't know or care about specific UI elements.
#
class QtHandler(logging.Handler):
    def __init__(self, slotfunc, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.signaller = Signaller()
        self.signaller.signal.connect(slotfunc)

    def emit(self, record):
        s = self.format(record)
        self.signaller.signal.emit(s, record)

#
# This example uses QThreads, which means that the threads at the Python level
# are named something like "Dummy-1". The function below gets the Qt name of the
# current thread.
#
def ctname():
    return QtCore.QThread.currentThread().objectName()

#
# Used to generate random levels for logging.
#
LEVELS = (logging.DEBUG, logging.INFO, logging.WARNING, logging.ERROR,
          logging.CRITICAL)

#
# This worker class represents work that is done in a thread separate to the
# main thread. The way the thread is kicked off to do work is via a button press
# that connects to a slot in the worker.
#

```

```

# Because the default threadName value in the LogRecord isn't much use, we add
# a qThreadName which contains the QThread name as computed above, and pass that
# value in an "extra" dictionary which is used to update the LogRecord with the
# QThread name.
#
# This example worker just outputs messages sequentially, interspersed with
# random delays of the order of a few seconds.
#
class Worker(QtCore.QObject):
    @Slot()
    def start(self):
        extra = {'qThreadName': ctname() }
        logger.debug('Started work', extra=extra)
        i = 1
        # Let the thread run until interrupted. This allows reasonably clean
        # thread termination.
        while not QtCore.QThread.currentThread().isInterruptionRequested():
            delay = 0.5 + random.random() * 2
            time.sleep(delay)
            try:
                if random.random() < 0.1:
                    raise ValueError('Exception raised: %d' % i)
                else:
                    level = random.choice(LEVELS)
                    logger.log(level, 'Message after delay of %3.1f: %d', delay, i,
↪ extra=extra)
            except ValueError as e:
                logger.exception('Failed: %s', e, extra=extra)
                i += 1

#
# Implement a simple UI for this cookbook example. This contains:
#
# * A read-only text edit window which holds formatted log messages
# * A button to start work and log stuff in a separate thread
# * A button to log something from the main thread
# * A button to clear the log window
#
class Window(QtWidgets.QWidget):

    COLORS = {
        logging.DEBUG: 'black',
        logging.INFO: 'blue',
        logging.WARNING: 'orange',
        logging.ERROR: 'red',
        logging.CRITICAL: 'purple',
    }

    def __init__(self, app):
        super().__init__()
        self.app = app
        self.textedit = te = QtWidgets.QPlainTextEdit(self)
        # Set whatever the default monospace font is for the platform
        f = QtGui.QFont('nosuchfont')
        if hasattr(f, 'Monospace'):
            f.setStyleHint(f.Monospace)

```

```

else:
    f.setStyleHint(f.StyleHint.Monospace) # for Qt6
    te.setFont(f)
    te.setReadOnly(True)
    PB = QtWidgets.QPushButton
    self.work_button = PB('Start background work', self)
    self.log_button = PB('Log a message at a random level', self)
    self.clear_button = PB('Clear log window', self)
    self.handler = h = QtHandler(self.update_status)
    # Remember to use qThreadName rather than threadName in the format string.
    fs = '%(asctime)s %(qThreadName)-12s %(levelname)-8s %(message)s'
    formatter = logging.Formatter(fs)
    h.setFormatter(formatter)
    logger.addHandler(h)
    # Set up to terminate the QThread when we exit
    app.aboutToQuit.connect(self.force_quit)

    # Lay out all the widgets
    layout = QtWidgets.QVBoxLayout(self)
    layout.addWidget(te)
    layout.addWidget(self.work_button)
    layout.addWidget(self.log_button)
    layout.addWidget(self.clear_button)
    self.setFixedSize(900, 400)

    # Connect the non-worker slots and signals
    self.log_button.clicked.connect(self.manual_update)
    self.clear_button.clicked.connect(self.clear_display)

    # Start a new worker thread and connect the slots for the worker
    self.start_thread()
    self.work_button.clicked.connect(self.worker.start)
    # Once started, the button should be disabled
    self.work_button.clicked.connect(lambda : self.work_button.
→setEnabled(False))

def start_thread(self):
    self.worker = Worker()
    self.worker_thread = QtCore.QThread()
    self.worker.setObjectName('Worker')
    self.worker_thread.setObjectName('WorkerThread') # for qThreadName
    self.worker.moveToThread(self.worker_thread)
    # This will start an event loop in the worker thread
    self.worker_thread.start()

def kill_thread(self):
    # Just tell the worker to stop, then tell it to quit and wait for that
    # to happen
    self.worker_thread.requestInterruption()
    if self.worker_thread.isRunning():
        self.worker_thread.quit()
        self.worker_thread.wait()
    else:
        print('worker has already exited.')

def force_quit(self):

```

```

    # For use when the window is closed
    if self.worker_thread.isRunning():
        self.kill_thread()

    # The functions below update the UI and run in the main thread because
    # that's where the slots are set up

    @Slot(str, logging.LogRecord)
    def update_status(self, status, record):
        color = self.COLORS.get(record.levelno, 'black')
        s = '<pre><font color="%s">%s</font></pre>' % (color, status)
        self.textedit.appendHtml(s)

    @Slot()
    def manual_update(self):
        # This function uses the formatted message passed in, but also uses
        # information from the record to format the message in an appropriate
        # color according to its severity (level).
        level = random.choice(LEVELS)
        extra = {'qThreadName': ctname()}
        logger.log(level, 'Manually logged!', extra=extra)

    @Slot()
    def clear_display(self):
        self.textedit.clear()

def main():
    QtCore.QThread.currentThread().setObjectName('MainThread')
    logging.getLogger().setLevel(logging.DEBUG)
    app = QtWidgets.QApplication(sys.argv)
    example = Window(app)
    example.show()
    if hasattr(app, 'exec'):
        rc = app.exec()
    else:
        rc = app.exec_()
    sys.exit(rc)

if __name__ == '__main__':
    main()

```

34 Logging to syslog with RFC5424 support

Although **RFC 5424** dates from 2009, most syslog servers are configured by default to use the older **RFC 3164**, which hails from 2001. When `logging` was added to Python in 2003, it supported the earlier (and only existing) protocol at the time. Since RFC5424 came out, as there has not been widespread deployment of it in syslog servers, the `SysLogHandler` functionality has not been updated.

RFC 5424 contains some useful features such as support for structured data, and if you need to be able to log to a syslog server with support for it, you can do so with a subclassed handler which looks something like this:

```

import datetime
import logging.handlers
import re
import socket

```

```

import time

class SysLogHandler5424(logging.handlers.SysLogHandler):

    tz_offset = re.compile(r'([+-]\d{2}) (\d{2})$')
    escaped = re.compile(r'([\\"\\])')

    def __init__(self, *args, **kwargs):
        self.msgid = kwargs.pop('msgid', None)
        self.appname = kwargs.pop('appname', None)
        super().__init__(*args, **kwargs)

    def format(self, record):
        version = 1
        asctime = datetime.datetime.fromtimestamp(record.created).isoformat()
        m = self.tz_offset.match(time.strftime('%z'))
        has_offset = False
        if m and time.timezone:
            hrs, mins = m.groups()
            if int(hrs) or int(mins):
                has_offset = True
        if not has_offset:
            asctime += 'Z'
        else:
            asctime += f'{hrs}:{mins}'
        try:
            hostname = socket.gethostname()
        except Exception:
            hostname = '-'
        appname = self.appname or '-'
        procid = record.process
        msgid = '-'
        msg = super().format(record)
        sdata = '-'
        if hasattr(record, 'structured_data'):
            sd = record.structured_data
            # This should be a dict where the keys are SD-ID and the value is a
            # dict mapping PARAM-NAME to PARAM-VALUE (refer to the RFC for what
            ↪these
            # mean)
            # There's no error checking here - it's purely for illustration, and
            ↪you
            # can adapt this code for use in production environments
            parts = []

            def replacer(m):
                g = m.groups()
                return '\\\ ' + g[0]

            for sdid, dv in sd.items():
                part = f'[{sdid}]'
                for k, v in dv.items():
                    s = str(v)
                    s = self.escaped.sub(replacer, s)
                    part += f' {k}="{s}"'
                part += ']'

```

```

        parts.append(part)
        sdata = ''.join(parts)
        return f'{version} {asctime} {hostname} {appname} {procid} {msgid} {sdata}
→ {msg}'

```

You'll need to be familiar with RFC 5424 to fully understand the above code, and it may be that you have slightly different needs (e.g. for how you pass structural data to the log). Nevertheless, the above should be adaptable to your specific needs. With the above handler, you'd pass structured data using something like this:

```

sd = {
    'foo@12345': {'bar': 'baz', 'baz': 'bozz', 'fizz': r'buzz'},
    'foo@54321': {'rab': 'baz', 'zab': 'bozz', 'zzif': r'buzz'}
}
extra = {'structured_data': sd}
i = 1
logger.debug('Message %d', i, extra=extra)

```

35 How to treat a logger like an output stream

Sometimes, you need to interface to a third-party API which expects a file-like object to write to, but you want to direct the API's output to a logger. You can do this using a class which wraps a logger with a file-like API. Here's a short script illustrating such a class:

```

import logging

class LoggerWriter:
    def __init__(self, logger, level):
        self.logger = logger
        self.level = level

    def write(self, message):
        if message != '\n': # avoid printing bare newlines, if you like
            self.logger.log(self.level, message)

    def flush(self):
        # doesn't actually do anything, but might be expected of a file-like
        # object - so optional depending on your situation
        pass

    def close(self):
        # doesn't actually do anything, but might be expected of a file-like
        # object - so optional depending on your situation. You might want
        # to set a flag so that later calls to write raise an exception
        pass

def main():
    logging.basicConfig(level=logging.DEBUG)
    logger = logging.getLogger('demo')
    info_fp = LoggerWriter(logger, logging.INFO)
    debug_fp = LoggerWriter(logger, logging.DEBUG)
    print('An INFO message', file=info_fp)
    print('A DEBUG message', file=debug_fp)

if __name__ == "__main__":
    main()

```

When this script is run, it prints

```
INFO:demo:An INFO message
DEBUG:demo:A DEBUG message
```

You could also use `LoggerWriter` to redirect `sys.stdout` and `sys.stderr` by doing something like this:

```
import sys

sys.stdout = LoggerWriter(logger, logging.INFO)
sys.stderr = LoggerWriter(logger, logging.WARNING)
```

You should do this *after* configuring logging for your needs. In the above example, the `basicConfig()` call does this (using the `sys.stderr` value *before* it is overwritten by a `LoggerWriter` instance). Then, you'd get this kind of result:

```
>>> print('Foo')
INFO:demo:Foo
>>> print('Bar', file=sys.stderr)
WARNING:demo:Bar
>>>
```

Of course, the examples above show output according to the format used by `basicConfig()`, but you can use a different formatter when you configure logging.

Note that with the above scheme, you are somewhat at the mercy of buffering and the sequence of write calls which you are intercepting. For example, with the definition of `LoggerWriter` above, if you have the snippet

```
sys.stderr = LoggerWriter(logger, logging.WARNING)
1 / 0
```

then running the script results in

```
WARNING:demo:Traceback (most recent call last):

WARNING:demo:  File "/home/runner/cookbook-loggerwriter/test.py", line 53, in
-><module>

WARNING:demo:
WARNING:demo:main()
WARNING:demo:  File "/home/runner/cookbook-loggerwriter/test.py", line 49, in main

WARNING:demo:
WARNING:demo:1 / 0
WARNING:demo:ZeroDivisionError
WARNING:demo::
WARNING:demo:division by zero
```

As you can see, this output isn't ideal. That's because the underlying code which writes to `sys.stderr` makes multiple writes, each of which results in a separate logged line (for example, the last three lines above). To get around this problem, you need to buffer things and only output log lines when newlines are seen. Let's use a slightly better implementation of `LoggerWriter`:

```
class BufferingLoggerWriter(LoggerWriter):
    def __init__(self, logger, level):
        super().__init__(logger, level)
        self.buffer = ''

    def write(self, message):
```

(다음 페이지에 계속)

```

if '\n' not in message:
    self.buffer += message
else:
    parts = message.split('\n')
    if self.buffer:
        s = self.buffer + parts.pop(0)
        self.logger.log(self.level, s)
    self.buffer = parts.pop()
    for part in parts:
        self.logger.log(self.level, part)

```

This just buffers up stuff until a newline is seen, and then logs complete lines. With this approach, you get better output:

```

WARNING:demo:Traceback (most recent call last):
WARNING:demo:  File "/home/runner/cookbook-loggerwriter/main.py", line 55, in
↪<module>
WARNING:demo:    main()
WARNING:demo:  File "/home/runner/cookbook-loggerwriter/main.py", line 52, in main
WARNING:demo:    1/0
WARNING:demo:ZeroDivisionError: division by zero

```

36 Patterns to avoid

Although the preceding sections have described ways of doing things you might need to do or deal with, it is worth mentioning some usage patterns which are *unhelpful*, and which should therefore be avoided in most cases. The following sections are in no particular order.

36.1 Opening the same log file multiple times

On Windows, you will generally not be able to open the same file multiple times as this will lead to a “file is in use by another process” error. However, on POSIX platforms you’ll not get any errors if you open the same file multiple times. This could be done accidentally, for example by:

- Adding a file handler more than once which references the same file (e.g. by a copy/paste/forget-to-change error).
- Opening two files that look different, as they have different names, but are the same because one is a symbolic link to the other.
- Forking a process, following which both parent and child have a reference to the same file. This might be through use of the `multiprocessing` module, for example.

Opening a file multiple times might *appear* to work most of the time, but can lead to a number of problems in practice:

- Logging output can be garbled because multiple threads or processes try to write to the same file. Although logging guards against concurrent use of the same handler instance by multiple threads, there is no such protection if concurrent writes are attempted by two different threads using two different handler instances which happen to point to the same file.
- An attempt to delete a file (e.g. during file rotation) silently fails, because there is another reference pointing to it. This can lead to confusion and wasted debugging time - log entries end up in unexpected places, or are lost altogether. Or a file that was supposed to be moved remains in place, and grows in size unexpectedly despite size-based rotation being supposedly in place.

Use the techniques outlined in [여러 프로세스에서 단일 파일에 로깅 하기](#) to circumvent such issues.

36.2 Using loggers as attributes in a class or passing them as parameters

While there might be unusual cases where you'll need to do this, in general there is no point because loggers are singletons. Code can always access a given logger instance by name using `logging.getLogger(name)`, so passing instances around and holding them as instance attributes is pointless. Note that in other languages such as Java and C#, loggers are often static class attributes. However, this pattern doesn't make sense in Python, where the module (and not the class) is the unit of software decomposition.

36.3 Adding handlers other than `NullHandler` to a logger in a library

Configuring logging by adding handlers, formatters and filters is the responsibility of the application developer, not the library developer. If you are maintaining a library, ensure that you don't add handlers to any of your loggers other than a `NullHandler` instance.

36.4 Creating a lot of loggers

Loggers are singletons that are never freed during a script execution, and so creating lots of loggers will use up memory which can't then be freed. Rather than create a logger per e.g. file processed or network connection made, use the *existing mechanisms* for passing contextual information into your logs and restrict the loggers created to those describing areas within your application (generally modules, but occasionally slightly more fine-grained than that).

37 Other resources

➔ 더 보기

모듈 `logging`

logging 모듈에 대한 API 레퍼런스

모듈 `logging.config`

logging 모듈용 구성 API.

모듈 `logging.handlers`

logging 모듈에 포함된 유용한 처리기.

Basic Tutorial

Advanced Tutorial

색인

R

RFC

RFC 3164, 62

RFC 5424, 41, 62

RFC 5424 Section 6, 41