로깅 HOWTO

출시 버전 ***3.9.19***

Guido van Rossum
and the Python development team

5월 01, 2024

Contents

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

1 기초 로깅 자습서

로깅은 어떤 소프트웨어가 실행될 때 발생하는 이벤트를 추적하는 수단입니다. 소프트웨어 개발자는 코드에 로깅 호출을 추가하여 특정 이벤트가 발생했음을 나타냅니다. 이벤트는 선택적으로 가변 데이터 (즉, 이벤트 발생마다 잠재적으로 다른 데이터)를 포함할 수 있는 설명 메시지로 기술됩니다. 이벤트는 또한 개발자가 이 벤트에 부여한 중요도를 가지고 있습니다; 중요도는 수준*(level)* 또는 심각도*(severity)* 라고도 부를 수 있습니다.

1.1 로깅을 사용할 때

로깅은 간단한 로깅 사용을 위한 일련의 편리 함수를 제공합니다. `debug()`, `info()`, `warning()`, `error()` 그리고 `critical()` 입니다. 로깅을 사용할 때를 결정하려면 아래 표를 참조하십시오. 일반적인 작업 집합 각각에 대해 사용하기에 가장 적합한 도구를 설명합니다.

수행하려는 작업	작업을 위한 최상의 도구
명령행 스크립트 또는 프로그램의 일반적인 사용을 위한 콘솔 출력 표시	`print()`
프로그램의 정상 작동 중에 발생하는 이벤트 보고 (가령 상태 모니터링이나 결함 조사)	`logging.info()` (또는 진단 목적의 아주 자세한 출력의 경우 `logging.debug()`)
특정 실행시간 이벤트와 관련하여 경고를 발행	라이브러리 코드에서 `warnings.warn()`: 문제를 피할 수 있고 경고를 제거하기 위해 클라이언트 응용 프로그램이 수정되어야 하는 경우 `logging.warning()`: 클라이언트 응용 프로그램이 할 수 있는 일이 없는 상황이지만 이벤트를 계속 주목해야 하는 경우
특정 실행시간 이벤트와 관련하여 에러를 보고	예외를 일으킵니다
예외를 발생시키지 않고 에러의 억제를 보고 (가령 장기 실행 서버 프로세스의 에러 처리기)	구체적인 에러와 응용 프로그램 영역에 적절한 `logging.error()`, `logging.exception()`, `logging.critical()`

로깅 함수는 추적되는 이벤트의 수준 또는 심각도를 따라 명명됩니다. 표준 수준과 그 용도는 아래에 설명되어 있습니다 (심각도가 높아지는 순서대로):

수준	사용할 때
`DEBUG`	상세한 정보. 보통 문제를 진단할 때만 필요합니다.
`INFO`	예상대로 작동하는지에 대한 확인.
`WARNING`	예상치 못한 일이 발생했거나 가까운 미래에 발생할 문제 (예를 들어 '디스크 공간 부족') 에 대한 표시. 소프트웨어는 여전히 예상대로 작동합니다.
`ERROR`	더욱 심각한 문제로 인해, 소프트웨어가 일부 기능을 수행하지 못했습니다.
`CRITICAL`	심각한 에러. 프로그램 자체가 계속 실행되지 않을 수 있음을 나타냅니다.

기본 수준은 `WARNING` 입니다. 이는 logging 패키지가 달리 구성되지 않는 한, 이 수준 이상의 이벤트만 추적된다는 것을 의미합니다.

추적되는 이벤트는 여러 방식으로 처리될 수 있습니다. 추적된 이벤트를 처리하는 가장 간단한 방법은 콘솔에 인쇄하는 것입니다. 또 다른 일반적인 방법은 디스크 파일에 기록하는 것입니다.

1.2 간단한 예

아주 간단한 예는 이렇습니다:

```
import logging
logging.warning('Watch out!')  # will print a message to the console
logging.info('I told you so')  # will not print anything
```

이 줄들을 스크립트에 입력하고 실행하면:

```
WARNING:root:Watch out!
```

이 콘솔에 출력됩니다. 기본 수준이 WARNING 이므로, INFO 메시지는 나타나지 않습니다. 인쇄된 메시지에는 수준 표시와 로깅 호출에 제공된 이벤트의 설명(즉, 'Watch out!') 이 포함됩니다. 당장은 'root' 부분에 대해서는 걱정하지 마십시오: 나중에 설명합니다. 필요한 경우 실제 출력을 매우 유연하게 포맷 할 수 있습니다; 포매팅 옵션도 나중에 설명합니다.

1.3 파일에 로깅 하기

매우 일반적인 상황은 로깅 이벤트를 파일에 기록하는 것이므로, 다음으로 살펴보겠습니다. 새로 시작된 파이썬 인터프리터에서 다음을 시도해보고, 위에서 설명한 세션을 계속 진행하지는 마십시오:

```
import logging
logging.basicConfig(filename='example.log', encoding='utf-8', level=logging.DEBUG)
logging.debug('This message should go to the log file')
logging.info('So should this')
logging.warning('And this, too')
logging.error('And non-ASCII stuff, too, like Øresund and Malmö')
```

버전 3.9에서 변경: *encoding* 인자가 추가되었습니다. 이전 파이썬 버전에서, 또는 지정되지 않으면, 사용된 인코딩은 open() 에서 사용되는 기본값입니다. 위의 예제에는 표시되지 않았지만, 이제 *errors* 인자를 전달하여 인코딩 에러 처리 방법을 결정할 수 있습니다. 사용 가능한 값과 기본값은, open() 설명서를 참조하십시오.

이제 파일을 열고 내용을 살펴본다면, 로그 메시지를 찾을 수 있습니다:

```
DEBUG:root:This message should go to the log file
INFO:root:So should this
WARNING:root:And this, too
ERROR:root:And non-ASCII stuff, too, like Øresund and Malmö
```

이 예제는 추적 임곗값 역할을 하는 로깅 수준을 설정하는 방법도 보여줍니다. 이 경우 임곗값을 DEBUG 로 설정했기 때문에 모든 메시지가 출력되었습니다.

다음과 같은 방식으로 명령행 옵션에서 로깅 수준을 설정하려고 하고:

```
--log=INFO
```

어떤 변수 *loglevel* 에 --log 로 전달된 매개 변수의 값이 들어있다면, 이런 방법으로:

```
getattr(logging, loglevel.upper())
```

level 인자를 통해 basicConfig() 에게 전달할 값을 얻습니다. 아마 다음 예제에서와같이, 사용자 입력 값을 오류 검사 하고 싶을 겁니다:

```
# assuming loglevel is bound to the string value obtained from the
# command line argument. Convert to upper case to allow the user to
# specify --log=DEBUG or --log=debug
numeric_level = getattr(logging, loglevel.upper(), None)
if not isinstance(numeric_level, int):
    raise ValueError('Invalid log level: %s' % loglevel)
logging.basicConfig(level=numeric_level, ...)
```

basicConfig() 에 대한 호출은 debug(), info() 등의 호출 전에 올 필요가 있습니다. 일회용의 간단한 설정 기능으로 의도되었기 때문에, 오직 첫 번째 호출만 뭔가 할 수 있습니다: 후속 호출은 사실상 아무 작업도 수행하지 않습니다.

위의 스크립트를 여러 번 실행하면, 후속 실행의 메시지가 *example.log* 파일에 추가됩니다. 이전 실행의 메시지를 기억하지 않고, 각 실행이 새로 시작하게 하려면, 위 예제에서 호출이 *filemode* 인자를 지정하도록 다음과 같이 변경할 수 있습니다:

```
logging.basicConfig(filename='example.log', filemode='w', level=logging.DEBUG)
```

출력은 이전과 같지만, 더는 로그 파일에 덧붙여지지 않으므로 이전 실행의 메시지는 손실됩니다.

1.4 여러 모듈에서의 로깅

프로그램이 여러 모듈로 구성되어있는 경우, 로깅을 구성하는 방법의 예는 다음과 같습니다:

```
# myapp.py
import logging
import mylib

def main():
    logging.basicConfig(filename='myapp.log', level=logging.INFO)
    logging.info('Started')
    mylib.do_something()
    logging.info('Finished')

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

```
# mylib.py
import logging

def do_something():
    logging.info('Doing something')
```

myapp.py 를 실행하면, *myapp.log* 에 다음과 같이 표시됩니다:

```
INFO:root:Started
INFO:root:Doing something
INFO:root:Finished
```

여러분이 기대한 것이기를 바랍니다. *mylib.py* 의 패턴을 사용하여 이것을 여러 모듈로 일반화 할 수 있습니다. 이 간단한 사용 패턴의 경우, 이벤트 설명을 보는 것 외에, 로그 파일을 들여다보는 것만으로는 메시지가 응용 프로그램의 어디서 왔는지 알 수 없습니다. 메시지의 위치를 추적하려면, 자습서 수준 이상의 문서를 참조해야 합니다 – 고급 로깅 자습서를 참조하세요.

1.5 변수 데이터 로깅

변수 데이터를 기록하려면, 이벤트 설명 메시지에 포맷 문자열을 사용하고 변수 데이터를 인자로 추가하십시오. 예를 들면:

```
import logging
logging.warning('%s before you %s', 'Look', 'leap!')
```

이것은 다음과 같이 출력합니다:

```
WARNING:root:Look before you leap!
```

보시다시피, 이벤트 설명 메시지에 가변 데이터를 병합하는데 이전 %-스타일의 문자열 포매팅을 사용합니다. 이전 버전과의 호환성을 위한 것입니다: logging 패키지는 `str.format()` 과 `string.Template` 과 같은 새로운 포매팅 옵션 이전부터 존재해왔습니다. 이 새로운 포매팅 옵션 역시 지원되지만, 이 자습서의 범위를 벗어납니다: 좀 더 자세한 정보는 formatting-styles를 참조하세요.

1.6 표시된 메시지의 포맷 변경

메시지를 표시하는 데 사용되는 포맷을 변경하려면 사용할 format을 지정해야 합니다:

```
import logging
logging.basicConfig(format='%(levelname)s:%(message)s', level=logging.DEBUG)
logging.debug('This message should appear on the console')
logging.info('So should this')
logging.warning('And this, too')
```

이것은 다음과 같이 인쇄합니다:

```
DEBUG:This message should appear on the console
INFO:So should this
WARNING:And this, too
```

앞의 예제에서 나타난 'root' 가 사라졌음에 주목하십시오. 포맷 문자열에 나타날 수 있는 모든 것은 logrecord-attributes 문서를 참고하게요. 하지만, 간단한 사용을 위해서는 *levelname* (심각도), *message* (이벤트 설명, 변수 데이터 포함) 와 아마도 발생 시각을 표시해야 할 것입니다. 이것은 다음 섹션에서 설명합니다.

1.7 메시지에 날짜/시간 표시

이벤트의 날짜와 시간을 표시하려면, 포맷 문자열에 '%(asctime)s' 을 넣으십시오:

```
import logging
logging.basicConfig(format='%(asctime)s %(message)s')
logging.warning('is when this event was logged.')
```

이것은 이런 식으로 인쇄합니다:

```
2010-12-12 11:41:42,612 is when this event was logged.
```

(위에 나온) 날짜/시간 표시의 기본 포맷은 ISO8601 또는 **RFC 3339**와 같습니다. 날짜/시간의 포맷을 좀 더 제어해야 하는 경우, 이 예제에서와같이 `basicConfig` 에 *datefmt* 인자를 제공하십시오:

```
import logging
logging.basicConfig(format='%(asctime)s %(message)s', datefmt='%m/%d/%Y %I:%M:%S %p')
logging.warning('is when this event was logged.')
```

그러면 다음과 같이 표시됩니다:

```
12/12/2010 11:46:36 AM is when this event was logged.
```

datefmt 인자의 형식은 `time.strftime()` 에 의해 지원되는 것과 같습니다.

1.8 다음 단계

이것으로 기본 자습서를 마칩니다. 여러분이 로깅을 시작하고 사용하는데 충분할 겁니다. logging 패키지가 더 많은 것들을 제공하지만, 최선의 결과를 얻으려면 다음 섹션을 읽는 데 시간을 조금 더 투자하시기 바랍니다. 준비되었다면, 좋아하는 음료 한잔 준비하시고 계속합시다.

로깅 요구가 간단하면, 위의 예제를 사용하여 자신의 스크립트에 로깅을 통합하십시오. 문제가 발생하거나 이해할 수 없는 부분이 있으면 comp.lang.python 유즈넷 그룹에 질문을 올리십시오 (https://groups.google.com/forum/#!forum/comp.lang.python 에 있습니다). 도움을 받는 데 아주 오래 걸리지는 않을 겁니다.

아직 계신가요? 위의 기본 섹션보다 약간 더 고급/심층적인 자습서를 제공하는 다음 몇 섹션을 계속 읽을 수 있습니다. 그 후에, logging-cookbook을 살펴볼 수 있습니다.

2 고급 로깅 자습서

logging 라이브러리는 모듈 방식으로 구성되며, 로거, 처리기, 필터 및 포매터와 같은 여러 범주의 구성 요소를 제공합니다.

- 로거는 응용 프로그램 코드가 직접 사용하는 인터페이스를 노출합니다.
- 처리기는 (로거에 의해 만들어진) 로그 레코드를 적절한 목적지로 보냅니다.
- 필터는 출력할 로그 레코드를 결정하기 위한 보다 정밀한 기능을 제공합니다.
- 포매터는 최종 출력에서 로그 레코드의 배치를 지정합니다.

로그 이벤트 정보는 `LogRecord` 인스턴스를 통해 로거, 처리기, 필터 및 포매터 간에 전달됩니다.

로깅은 `Logger` 클래스(이하 로거 *(loggers)* 라고 합니다) 인스턴스의 메서드를 호출하여 수행됩니다. 각 인스턴스에는 이름이 있으며, 점(마침표)을 구분 기호로 사용하여 개념적으로는 이름 공간 계층 구조로 배열됩니다. 예를 들어, 'scan' 이라는 로거는 'scan.text', 'scan.html' 및 'scan.pdf' 로거의 부모입니다. 로거 이름은 원하는 어떤 것이건 될 수 있으며, 로그 된 메시지가 시작된 응용 프로그램 영역을 나타냅니다.

로거의 이름을 지을 때 사용할 좋은 규칙은 다음과 같이 로깅을 사용하는 각 모듈에서 모듈 수준 로거를 사용하는 것입니다:

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

이것은 로거 이름이 패키지/모듈 계층을 추적한다는 것을 의미하며, 로거 이름으로부터 이벤트가 기록되는 위치를 직관적으로 명확히 알 수 있습니다.

로거 계층의 뿌리를 루트 로거라고 합니다. 이것이 `debug()`, `info()`, `warning()`, `error()` 그리고 `critical()` 함수에 의해 사용되는 로거인데, 루트 로거의 같은 이름의 메서드들 호출합니다. 함수와 메서드는 같은 서명을 가집니다. 루트 로거의 이름은 로그 된 출력에 'root' 로 인쇄됩니다.

물론 로그 메시지를 다른 대상에 기록하는 것도 가능합니다. 로그 메시지를 파일, HTTP GET/POST 위치, SMTP를 통한 전자 메일, 일반 소켓, 큐 또는 syslog 나 윈도우 NT 이벤트 로그와 같은 OS 특정 로깅 메커니즘에 쓰는 지원이 패키지에 포함되어 있습니다. 목적지는 처리기 *(handler)* 클래스에 의해 제공됩니다. 내장된 처리기 클래스에 의해 충족되지 않는 특별한 요구 사항이 있는 경우, 자체 로그 대상 클래스를 작성할 수 있습니다.

기본적으로, 로그 메시지에는 대상이 설정되지 않습니다. 자습서 예제에서와같이 `basicConfig()`를 사용하여 대상(가령 콘솔 또는 파일)을 지정할 수 있습니다. `debug()`, `info()`, `warning()`, `error()` 및

`critical()` 함수를 호출하면 목적지가 설정되어 있지 않은지 확인합니다; 설정되지 않았다면, 실제 메시지 출력을 하기 위해 루트 로거에 위임하기 전에, 콘솔(`sys.stderr`)을 대상으로 설정하고 표시되는 메시지의 기본 포맷을 설정합니다.

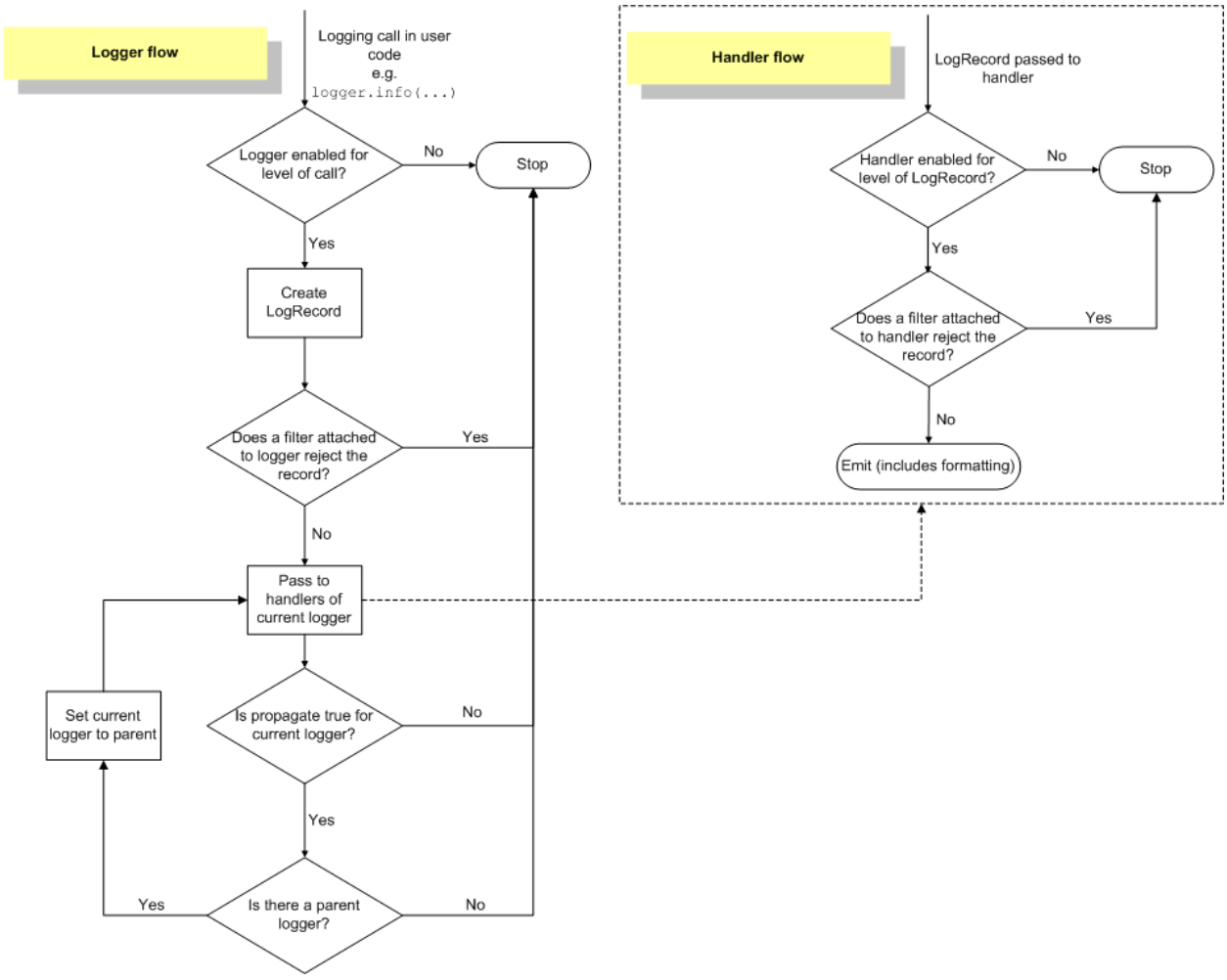
메시지에 대해 `basicConfig()` 에 의해 설정된 기본 포맷은 다음과 같습니다:

```
severity:logger name:message
```

포맷 문자열을 *format* 키워드 인자로 `basicConfig()` 에 전달하여 이를 변경할 수 있습니다. 포맷 문자열 작성 방법과 관련된 모든 옵션은 formatter-objects를 참조하십시오.

2.1 로깅 흐름

로거 및 처리기에서 로그 이벤트 정보의 흐름은 다음 도표에 설명되어 있습니다.



2.2 로거

Logger 객체는 세 가지 작업을 합니다. 첫째, 응용 프로그램이 실행시간에 메시지를 기록 할 수 있도록 여러 메서드를 응용 프로그램 코드에 노출합니다. 둘째, 로거 객체는 심각도 (기본 필터링 장치) 또는 필터 객체에 따라 어떤 로그 메시지를 처리할지 결정합니다. 셋째, 로거 객체는 관련 로그 메시지를 관심 있는 모든 로그 처리기로 전달합니다.

로거 객체에서 가장 널리 사용되는 메서드는 두 가지 범주로 분류됩니다: 구성 및 메시지 전송

다음은 가장 일반적인 구성 메서드입니다:

- Logger.setLevel() 은 로거가 처리할 가장 낮은 심각도의 로그 메시지를 지정합니다. debug은 가장 낮은 내장 심각도 수준이고 critical은 가장 높은 내장 심각도입니다. 예를 들어, 심각도 수준이 INFO이면 로거는 INFO, WARNING, ERROR 및 CRITICAL 메시지만 처리하고 DEBUG 메시지는 무시합니다.
- Logger.addHandler() 와 Logger.removeHandler() 는 로거 객체에서 처리기 객체를 추가하고 제거합니다. 처리기는 처리기에서 더욱 자세히 다룹니다.
- Logger.addFilter() 와 Logger.removeFilter() 는 로거 객체에서 필터 객체를 추가하고 제거합니다. 필터는 filter에서 더욱 자세히 다룹니다.

생성 한 모든 로거에서 항상 이 메서드를 호출할 필요는 없습니다. 이 섹션의 마지막 두 단락을 참조하십시오.

로거 객체가 구성된 상태에서 다음 메서드는 로그 메시지를 만듭니다:

- Logger.debug(), Logger.info(), Logger.warning(), Logger.error() 그리고 Logger.critical() 은 모두 메시지 와 메서드 이름에 해당하는 수준으로 로그 레코드를 만듭니다. 메시지는 실제로는 포맷 문자열이며, %s, %d, %f 등의 표준 문자열 치환 문법을 포함 할 수 있습니다. 나머지 인자들은 메시지의 치환 필드에 해당하는 객체 목록입니다. **kwargs 의 경우, 로깅 메서드는 exc_info 라는 키워드에 대해서만 신경을 쓰고 예외 정보를 로깅 할지를 결정할 때 사용합니다.
- Logger.exception() 은 Logger.error() 와 비슷한 로그 메시지를 생성합니다. 차이점은 Logger.exception() 가 스택 트레이스를 덤프한다는 것입니다. 예외 처리기에서만 이 메서드를 호출하십시오.
- Logger.log() 는 명시적 인자로 로그 수준을 받아들입니다. 위에 나열된 로그 수준 편의 메서드를 사용하는 것보다 로깅 메시지를 만들 때 더 장황해지지만, 사용자 정의 로그 수준으로 로깅 하는 방법입니다.

getLogger() 는 이름이 제공되는 경우 지정된 이름을 가진 로거 인스턴스에 대한 참조를 반환하고, 그렇지 않으면 root 를 반환합니다. 이름은 마침표로 구분된 계층적 구조입니다. 같은 이름으로 getLogger() 를 여러 번 호출하면 같은 로거 객체에 대한 참조를 반환합니다. 계층적 목록에서 더 아래쪽에 있는 로거는 목록에서 상위에 있는 로거의 자식입니다. 예를 들어, 이름이 foo 인 로거가 주어지면, foo.bar, foo.bar.baz, 그리고 foo.bam 의 이름을 가진 로거는 모두 foo 의 자손입니다.

로거에는 실효 수준 이라는 개념이 있습니다. 수준이 로거에 명시적으로 설정되지 않은 경우, 부모 수준을 실효 수준으로 대신 사용합니다. 부모가 명시적 수준 집합을 가지고 있지 않으면, 다시 그것의 부모가 검사되고, 등등 - 명시적으로 설정된 수준이 발견될 때까지 모든 조상이 검색됩니다. 루트 로거는 항상 명시적인 수준 집합(기본적으로 WARNING)을 가지고 있습니다. 이벤트 처리 여부를 결정할 때, 로거의 실효 수준이 이벤트가 로거 처리기로 전달되는지를 판별하는 데 사용됩니다.

자식 로거는 조상 로거와 연관된 처리기로 메시지를 전달합니다. 이 때문에 응용 프로그램에서 사용하는 모든 로거에 대해 처리기를 정의하고 구성할 필요가 없습니다. 최상위 수준 로거에 대한 처리기를 구성하고 필요에 따라 자식 로거를 만드는 것으로 충분합니다. (그러나, 로거의 *propagate* 어트리뷰트를 False 로 설정하여 전파를 끌 수 있습니다.)

2.3 처리기

Handler 객체는 (로그 메시지의 심각도를 기반으로) 적절한 로그 메시지를 처리기의 지정된 대상으로 전달하는 역할을 합니다. Logger 객체는 addHandler() 메서드를 사용하여 0개 이상의 처리기 객체를 자신에게 추가 할 수 있습니다. 예를 들어, 응용 프로그램은 모든 로그 메시지를 로그 파일로 보내고, 에러(error)와 그 이상의 모든 로그 메시지를 표준 출력으로 보내고, 모든 심각한 에러(critical) 메시지를 전자 메일 주소로 보낼 수 있습니다. 이 시나리오에서는 각 처리기가 특정 심각도의 메시지를 특정 위치로 보내는 3개의 개별 처리기가 필요합니다.

표준 라이브러리에는 꽤 많은 처리기형이 포함되어있습니다 (유용한 처리기 참조). 자습서는 주로 StreamHandler 와 FileHandler 를 예제에서 사용합니다.

처리기에는 응용 프로그램 개발자가 직접 신경 써야 할 메서드가 거의 없습니다. 기본 제공 처리기 객체를 사용하는 (즉, 사용자 정의 처리기를 만들지 않는) 응용 프로그램 개발자와 관련이 있는 처리기 메서드는 다음과 같은 구성 메서드뿐입니다:

- setLevel() 메서드는 로거 객체에서와 마찬가지로 적절한 목적지로 보내지는 가장 낮은 심각도를 지정합니다. 왜 두 개의 setLevel() 메서드가 있어야 할까요? 로거에 설정된 수준은 처리기에 전달할 메시지의 심각도를 판별합니다. 각 처리기에 설정된 수준은 처리기가 전송할 메시지를 결정합니다.
- setFormatter() 는 처리기가 사용할 포매터 객체를 선택합니다.
- addFilter() 와 removeFilter() 는 각각 처리기에서 필터 객체를 구성하고 해제합니다.

응용 프로그램 코드는 Handler 의 인스턴스를 직접 인스턴스화해서 사용해서는 안 됩니다. 대신, Handler 클래스는 모든 처리기가 가져야 하는 인터페이스를 정의하고 자식 클래스가 사용할 수 있는 (또는 재정의할 수 있는) 기본 동작을 설정하는 베이스 클래스입니다.

2.4 포매터

포매터 객체는 로그 메시지의 최종 순서, 구조 및 내용을 구성합니다. 베이스 logging.Handler 클래스와는 달리, 응용 프로그램 코드는 포매터 클래스를 인스턴스화 할 수 있습니다. 응용 프로그램에 특별한 동작이 필요한 경우 포매터의 서브 클래스를 만들 수도 있습니다. 생성자는 세 가지 선택적 인자를 취합니다 – 메시지 포맷 문자열, 날짜 포맷 문자열 및 스타일 지시자.

logging.Formatter.**__init__**(*fmt=None*, *datefmt=None*, *style='%'*)

메시지 포맷 문자열이 없으면, 기본값은 날 메시지를 사용하는 것입니다. 날짜 포맷 문자열이 없으면, 기본 날짜 형식은 다음과 같습니다:

```
%Y-%m-%d %H:%M:%S
```

끝에 밀리 초가 기록됩니다. style 은 '%', '{' 또는 '$' 중 하나입니다. 이 중 하나가 지정되지 않으면, '%' 가 사용됩니다.

style 이 '%' 이면, 메시지 포맷 문자열은 %(<dictionary key>)s 스타일의 문자열 치환을 사용합니다; 가능한 키는 logrecord-attributes 에 문서로 만들어져 있습니다. style 이 '{' 인 경우 메시지 포맷 문자열은 str.format()(키워드 인자 사용)과 호환되는 것으로 가정하고, 스타일이 '$' 이면 메시지 포맷 문자열은 string.Template.substitute() 가 기대하는 것과 일치해야 합니다.

버전 3.2에서 변경: style 매개 변수를 추가했습니다.

다음 메시지 포맷 문자열은 사람이 읽을 수 있는 형식의 시간, 메시지의 심각도 및 메시지의 내용을 순서대로 기록합니다:

```
'%(asctime)s - %(levelname)s - %(message)s'
```

포매터는 사용자가 구성 할 수 있는 함수를 사용하여 레코드의 생성 시간을 튜플로 변환합니다. 기본적으로, time.localtime() 이 사용됩니다; 특정 포매터 인스턴스에 대해 이를 변경하려면, 인스턴스의 converter

어트리뷰트를 `time.localtime()` 또는 `time.gmtime()` 과 같은 서명을 가진 함수로 설정하십시오. 모든 포매터를 변경하려면, 예를 들어 모든 로깅 시간을 GMT로 표시하려면, Formatter 클래스의 `converter` 어트리뷰트를 설정하십시오 (GMT 표시를 위해 `time.gmtime` 으로).

2.5 로깅 구성

프로그래머는 세 가지 방법으로 로깅을 구성 할 수 있습니다:

1. 위에 나열된 구성 메서드를 호출하는 파이썬 코드를 사용하여 로거, 처리기 및 포매터를 명시적으로 만듭니다.
2. 로깅 구성 파일을 만들고, `fileConfig()` 함수를 사용하여 그것을 읽습니다.
3. 구성 정보의 딕셔너리를 만들고, `dictConfig()` 함수에 전달합니다.

마지막 두 옵션에 대한 참조 설명서는 logging-config-api를 참조하십시오. 다음 예제는 파이썬 코드를 사용하여 매우 단순한 로거, 콘솔 처리기 및 간단한 포매터를 구성합니다:

```
import logging

# create logger
logger = logging.getLogger('simple_example')
logger.setLevel(logging.DEBUG)

# create console handler and set level to debug
ch = logging.StreamHandler()
ch.setLevel(logging.DEBUG)

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

# add formatter to ch
ch.setFormatter(formatter)

# add ch to logger
logger.addHandler(ch)

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

명령행에서 이 모듈을 실행하면 다음과 같이 출력됩니다:

```
$ python simple_logging_module.py
2005-03-19 15:10:26,618 - simple_example - DEBUG - debug message
2005-03-19 15:10:26,620 - simple_example - INFO - info message
2005-03-19 15:10:26,695 - simple_example - WARNING - warn message
2005-03-19 15:10:26,697 - simple_example - ERROR - error message
2005-03-19 15:10:26,773 - simple_example - CRITICAL - critical message
```

다음의 파이썬 모듈은 위에 열거된 예제와 거의 같고 객체의 이름 만 다른 로거, 처리기 및 포매터를 생성합니다:

```
import logging
import logging.config

```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
logging.config.fileConfig('logging.conf')

# create logger
logger = logging.getLogger('simpleExample')

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

logging.conf 파일은 이렇습니다:

```
[loggers]
keys=root,simpleExample

[handlers]
keys=consoleHandler

[formatters]
keys=simpleFormatter

[logger_root]
level=DEBUG
handlers=consoleHandler

[logger_simpleExample]
level=DEBUG
handlers=consoleHandler
qualname=simpleExample
propagate=0

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

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

출력은 구성 파일 기반이 아닌 예제와 거의 같습니다:

```
$ python simple_logging_config.py
2005-03-19 15:38:55,977 - simpleExample - DEBUG - debug message
2005-03-19 15:38:55,979 - simpleExample - INFO - info message
2005-03-19 15:38:56,054 - simpleExample - WARNING - warn message
2005-03-19 15:38:56,055 - simpleExample - ERROR - error message
2005-03-19 15:38:56,130 - simpleExample - CRITICAL - critical message
```

구성 파일 접근법이 파이썬 코드 접근법보다 몇 가지 장점이 있음을 알 수 있습니다. 주로 구성과 코드의 분리와 코더가 아니라도 로깅 속성을 쉽게 수정할 수 있다는 것입니다.

경고: `fileConfig()` 함수는 기본 매개 변수 `disable_existing_loggers` 를 취하는데, 하위 호환성을 위해 기본값은 `True` 입니다. 이것들은 여러분이 원하는 것일 수도 있고 아닐 수도 있습니다. 왜냐하면 `fileConfig()` 호출 전에 존재하는 모든 비 루트 로거들이, 구성에서 명시적으로 명명되지 않는 한, 비활성화되기 때문입니다. 자세한 내용은 참조 설명서를 참조하고, 원한다면 이 매개 변수에 `False` 를 지정하십시오.

`dictConfig()` 에 전달된 딕셔너리 또한 `disable_existing_loggers` 키로 논리값을 지정할 수 있습니다. 딕셔너리에 명시적으로 지정되지 않으면 기본적으로 `True` 로 해석됩니다. 이것은 위에서 설명한 로거 비활성화 동작으로 이어지는데, 여러분이 원하는 것이 아닐 수도 있습니다. 이 경우 키에 명시적으로 `False` 값을 제공하십시오.

구성 파일에서 참조되는 클래스 이름은 logging 모듈에 상대적이거나, 일반적인 임포트 메커니즘을 사용하여 결정할 수 있는 절댓값이어야 합니다. 따라서, `WatchedFileHandler` (logging 모듈에 상대적) 또는 `mypackage.mymodule.MyHandler` (패키지 `mypackage` 와 모듈 `mymodule` 에 정의된 클래스, 여기서 `mypackage` 는 파이썬 임포트 경로에서 사용 가능해야 합니다).

파이썬 3.2에서는 구성 정보를 보관하는 딕셔너리를 사용하여 로깅을 구성하는 새로운 방법이 도입되었습니다. 이는 위에 설명된 구성 파일 기반 접근 방식의 기능을 제공하며, 새로운 응용 프로그램 및 배포에 권장되는 구성 방법입니다. 파이썬 딕셔너리가 구성 정보를 저장하는 데 사용되고, 다른 방법을 사용하여 해당 딕셔너리를 채울 수 있으므로 더 많은 구성 옵션을 갖게 됩니다. 예를 들어, 구성 딕셔너리를 채우는데 JSON 형식의 구성 파일이나, YAML 처리 기능에 액세스할 수 있는 경우, YAML 형식의 파일을 사용할 수 있습니다. 물론, 파이썬 코드로 딕셔너리를 만들거나, 소켓을 통해 피클 된 형태로 수신하거나, 그 밖의 응용 프로그램에 적합한 어떤 방식도 사용할 수 있습니다.

다음은 새로운 딕셔너리 기반 접근 방식으로 YAML 형식으로 위와 같이 구성한 예입니다:

```
version: 1
formatters:
  simple:
    format: '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
handlers:
  console:
    class: logging.StreamHandler
    level: DEBUG
    formatter: simple
    stream: ext://sys.stdout
loggers:
  simpleExample:
    level: DEBUG
    handlers: [console]
    propagate: no
root:
  level: DEBUG
  handlers: [console]
```

딕셔너리를 사용하여 로깅 하는 방법에 대한 자세한 내용은 logging-config-api를 참조하십시오.

2.6 구성이 제공되지 않으면 어떻게 되는가

로깅 구성이 제공되지 않으면, 로깅 이벤트를 출력해야 하지만 이벤트를 출력하는 처리기를 찾을 수 없는 상황이 발생할 수 있습니다. 이러한 상황에서 logging 패키지의 동작은 파이썬 버전에 따라 다릅니다.

3.2 이전의 파이썬 버전의 경우 동작은 다음과 같습니다:

- *logging.raiseExceptions* 가 `False` (프로덕션 모드) 이면, 이벤트가 조용히 무시됩니다.
- *logging.raiseExceptions* 가 `True` (개발 모드) 이면, 'No handlers could be found for logger X.Y.Z' 라는 메시지가 한 번 인쇄됩니다.

파이썬 3.2 및 이후 버전에서는 다음과 같이 동작합니다:

- 이 이벤트는 `logging.lastResort` 에 저장된 '최후 수단 처리기'를 사용하여 출력됩니다. 이 내부 처리기는 어떤 로거와도 연관되어 있지 않고, 이벤트 설명 메시지를 `sys.stderr` 의 현재 값으로 쓰는 (따라서 현재 효과를 발휘하고 있는 모든 리디렉션을 존중합니다) `StreamHandler` 처럼 동작합니다. 메시지에는 어떤 포매팅도 적용되지 않습니다 - 이벤트 설명 메시지가 그대로 인쇄됩니다. 처리기의 수준은 `WARNING` 으로 설정되어 있으므로, 이보다 크거나 같은 심각도의 모든 이벤트가 출력됩니다.

3.2 이전의 동작을 얻으려면, `logging.lastResort` 를 `None` 으로 설정할 수 있습니다.

2.7 라이브러리 로깅 구성

로깅을 사용하는 라이브러리를 개발할 때, 라이브러리에서 로깅을 사용하는 방법을 문서로 만들어야 합니다 - 예를 들어, 사용된 로거의 이름. 또한, 로깅 구성에 대한 고려가 필요합니다. 사용하는 응용 프로그램이 로깅을 사용하지 않고 라이브러리 코드가 로깅을 호출하면, (앞 절에서 설명했듯이) 심각도가 `WARNING` 이상인 이벤트는 `sys.stderr` 에 출력됩니다. 이것이 최상의 기본 동작인 것으로 간주합니다.

어떤 이유로 로깅 구성이 없을 때 이 메시지가 인쇄되는 것을 원하지 않는다면, 라이브러리의 최상위 로거에 아무것도 하지 않는 처리기를 연결할 수 있습니다. 이렇게 하면 메시지가 인쇄되는 것을 피할 수 있습니다. 라이브러리의 이벤트를 위한 처리기가 항상 존재하기 때문입니다: 단지 아무런 출력도 만들지 않을 뿐입니다. 라이브러리 사용자가 응용 프로그램이 사용하기 위해 로깅을 구성하면, 아마도 그 구성이 어떤 처리기를 추가할 것이고, 수준이 적절하게 구성된 경우 라이브러리에서 이루어진 로깅 호출은 그 처리기로 정상적으로 출력을 보낼 것입니다.

아무것도 하지 않는 처리기가 logging 패키지에 포함되어 있습니다: `NullHandler` (파이썬 3.1부터). 이 처리기의 인스턴스는 라이브러리가 사용하는 로깅 이름 공간의 최상위 로거에 추가될 수 있습니다 (만약 로깅 구성이 없는 경우 라이브러리가 로깅한 이벤트가 `sys.stderr` 에 출력되는 것을 막으려면). 라이브러리 *foo* 에 의한 모든 로깅이 'foo.x', 'foo.x.y' 등과 일치하는 이름을 가진 로거들만 사용한다면, 다음과 같은 코드가:

```
import logging
logging.getLogger('foo').addHandler(logging.NullHandler())
```

원하는 효과를 주어야 합니다. 만약 조직이 여러 라이브러리를 만든다면, 지정된 로거 이름으로 단순한 'foo' 대신에 'orgname.foo' 를 사용할 수 있습니다.

참고: `NullHandler` 이외의 처리기를 라이브러리의 로거에 추가하지 않는 것이 좋습니다. 이것은 처리기 구성이 라이브러리를 사용하는 응용 프로그램 개발자의 특권이기 때문입니다. 응용 프로그램 개발자는 사용자와 응용 프로그램에 가장 적합한 처리기가 무엇인지 알고 있습니다. 여러분이 처리기를 '이면에서' 추가하면, 단위 테스트를 수행하고 그들의 요구사항에 맞는 로그를 전달하는 작업을 방해할 수 있습니다.

---

3 로깅 수준

로깅 수준의 숫자 값은 다음 표에 나와 있습니다. 이것은 주로 여러분 자신의 수준을 정의하고 사전 정의된 수준에 상대적인 특정 값을 갖도록 하려는 경우 관심의 대상입니다. 같은 숫자 값을 가진 수준을 정의하면 미리 정의된 값을 덮어씁니다; 사전 정의된 이름이 유실됩니다.

수준	숫자 값
CRITICAL	50
ERROR	40
WARNING	30
INFO	20
DEBUG	10
NOTSET	0

또한, 수준은 개발자가 설정하거나 저장된 로깅 구성을 로드하여 로거와 연관될 수 있습니다. 로깅 메서드가 로거에 호출되면, 로거는 자체 수준을 메서드 호출과 연관된 수준과 비교합니다. 로거 수준이 메서드 호출보다 높으면 실제로 로깅 메시지가 생성되지 않습니다. 이것은 로그 출력의 상세도를 제어하는 기본 메커니즘입니다.

로깅 메시지는 `LogRecord` 클래스의 인스턴스로 인코딩됩니다. 로거가 실제로 이벤트를 로그 하기로 하면, `LogRecord` 인스턴스가 로깅 메시지로부터 만들어집니다.

로깅 메시지는 `Handler` 클래스의 서브 클래스의 인스턴스인 처리기 *(handlers)* 를 통한 전달 메커니즘을 적용받게 됩니다. 처리기는 로그된 메시지(`LogRecord` 형식)가 대상 독자(가령 최종 사용자, 지원 데스크 직원, 시스템 관리자, 개발자)에게 유용한 특정 위치(또는 위치 집합)에 도달하도록 할 책임이 있습니다. 처리기로는 특정 목적지를 위한 `LogRecord` 인스턴스가 전달됩니다. 각 로거에는 0개 이상의 처리기가 연관될 수 있습니다(`Logger` 의 `addHandler()` 메서드를 통해). 로거에 직접 연관된 모든 처리기뿐만 아니라, (로거의 *propagate* 플래그가 거짓 값으로 설정되어 있지 않은 한, 그 지점에서 부모 처리기로의 전달이 멈춥니다) 로거의 모든 조상과 연관된 모든 처리기가 메시지를 전달하도록 호출됩니다.

로거와 마찬가지로, 처리기도 연관된 수준을 가질 수 있습니다. 처리기의 수준은 로거의 수준과 같은 방식으로 필터의 역할을 합니다. 처리기가 실제로 이벤트를 전달하기로 하면, `emit()` 메서드가 목적지로 메시지를 보내기 위해 사용됩니다. `Handler` 의 대부분의 사용자 정의 서브 클래스는 이 `emit()` 를 재정의해야 합니다.

3.1 사용자 정의 수준

여러분 자신의 수준을 정의하는 것이 가능하지만 필요하지는 않은데, 현재의 수준이 실무 경험에 근거하여 선택되었기 때문입니다. 하지만, 사용자 정의 수준이 필요하다고 확신하는 경우, 이를 수행할 때 많은 주의를 기울여야 합니다. 라이브러리를 개발하고 있다면 사용자 지정 수준을 정의하는 것은 매우 나쁜 생각일 수 있습니다. 왜냐하면, 여러 라이브러리 작성자가 모두 자신의 사용자 정의 수준을 정의하면, 주어진 숫자 값이 라이브러리마다 각기 다른 것을 의미 할 수 있으므로, 사용 중인 개발자가 함께 사용하는 여러 라이브러리의 로깅 결과를 제어하거나 해석하는 것이 어려울 수 있기 때문입니다.

4 유용한 처리기

베이스 Handler 클래스 외에도 많은 유용한 서브 클래스가 제공됩니다:

1. StreamHandler 인스턴스는 스트림(파일류 객체)에 메시지를 보냅니다.
2. FileHandler 인스턴스는 디스크 파일에 메시지를 보냅니다.
3. BaseRotatingHandler 는 특정 지점에서 로그 파일을 회전시키는 처리기의 베이스 클래스입니다. 직접 인스턴스화하는 것은 아닙니다. 대신 RotatingFileHandler 또는 TimedRotatingFileHandler 를 사용하십시오.
4. RotatingFileHandler 인스턴스는 디스크 파일에 메시지를 보내는데, 최대 로그 파일 크기와 로그 파일 회전을 지원합니다.
5. TimedRotatingFileHandler 인스턴스는 디스크 파일에 메시지를 보내는데, 일정한 시간 간격으로 로그 파일을 회전시킵니다.
6. SocketHandler 인스턴스는 TCP/IP 소켓에 메시지를 보냅니다. 3.4부터, 유닉스 도메인 소켓도 지원됩니다.
7. DatagramHandler 인스턴스는 UDP 소켓에 메시지를 보냅니다. 3.4부터, 유닉스 도메인 소켓도 지원됩니다.
8. SMTPHandler 인스턴스는 지정된 전자 우편 주소로 메시지를 보냅니다.
9. SysLogHandler 인스턴스는 유닉스 syslog 데몬(원격 기계에 있는 것도 가능합니다)에 메시지를 보냅니다.
10. NTEventLogHandler 인스턴스는 윈도우 NT/2000/XP 이벤트 로그에 메시지를 보냅니다.
11. MemoryHandler 인스턴스는 메모리에 있는 버퍼에 메시지를 보내는데, 특정 기준이 만족 될 때마다 플러시 됩니다.
12. HTTPHandler 인스턴스는 GET 또는 POST 을 사용해서 HTTP 서버에 메시지를 보냅니다.
13. WatchedFileHandler 인스턴스는 그들이 로깅하고 있는 파일을 감시합니다. 파일이 변경되면 닫히고 파일 이름을 사용하여 다시 열립니다. 이 처리기는 유닉스 계열 시스템에서만 유용합니다; 윈도우는 사용된 하부 메커니즘을 지원하지 않습니다.
14. QueueHandler 인스턴스는 queue 또는 multiprocessing 모듈에 구현된 것과 같은 큐로 메시지를 보냅니다.
15. NullHandler 인스턴스는 에러 메시지로 아무것도 하지 않습니다. 라이브러리 개발자가 로깅을 사용하지만, 라이브러리 사용자가 로깅을 구성하지 않으면 표시될 수 있는 'No handlers could be found for logger XXX' 라는 메시지를 피하려고 할 때 사용합니다. 자세한 정보는 라이브러리 로깅 구성 를 보십시오.

버전 3.1에 추가: NullHandler 클래스.

버전 3.2에 추가: QueueHandler 클래스.

NullHandler, StreamHandler 와 FileHandler 클래스는 코어 logging 패키지에 정의되어 있습니다. 다른 처리기는 하위 모듈인 logging.handlers 에 정의되어 있습니다. (구성 기능을 위한, 또 다른 하위 모듈 logging.config 도 있습니다.)

로그된 메시지는 Formatter 클래스의 인스턴스를 통해 표시를 위해 포맷됩니다. % 연산자와 딕셔너리와 함께 사용하기에 적합한 포맷 문자열로 초기화됩니다.

일괄 처리로 여러 개의 메시지를 포맷하려면, BufferingFormatter 의 인스턴스를 사용할 수 있습니다. 포맷 문자열(일괄 처리 때 각 메시지에 적용됩니다)에 더해, 헤더와 트레일러 포맷 문자열에 대한 고려가 있습니다.

로거 수준과 처리기 수준을 기반으로 필터링하는 것만으로는 충분하지 않은 경우, Logger 및 Handler 인스턴스에 (addFilter() 메서드를 통해) Filter 인스턴스를 추가 할 수 있습니다. 메시지를 더 처리하기로

하기 전에, 로거와 처리기는 모든 필터에 허락을 요청합니다. 한 필터라도 거짓 값을 반환하면 메시지는 더 처리되지 않습니다.

기본적인 `Filter` 기능은 특정 로거 이름으로 필터링하는 것을 지원합니다. 이 기능을 사용하면, 명명된 로거와 그 자식으로 보낸 메시지는 필터를 통과하도록 허용되고, 다른 모든 메시지는 삭제됩니다.

5 로깅 중에 발생하는 예외

logging 패키지는 프로덕션에서 로깅 하는 동안 발생하는 예외를 삼키도록 설계되었습니다. 이는 로깅 이벤트를 처리하는 동안 발생하는 에러(가령 잘못된 로깅 구성, 네트워크 또는 기타 유사한 에러)가 로깅을 사용하는 응용 프로그램을 조기에 종료시키지 않도록 하기 위한 것입니다.

`SystemExit` 과 `KeyboardInterrupt` 예외는 절대 삼켜지지 않습니다. `Handler` 서브 클래스의 `emit()` 메서드가 실행되는 동안 발생하는 다른 예외는 `handleError()` 메서드에 전달됩니다.

`Handler` 에 있는 `handleError()` 의 기본 구현은 모듈 수준 변수 `raiseExceptions` 가 설정되어 있는지를 검사합니다. 설정되어있으면, 트레이스백이 `sys.stderr` 에 인쇄됩니다. 설정되지 않았으면 예외를 삼킵니다.

참고: `raiseExceptions` 의 기본값은 `True` 입니다. 개발 중에는, 보통 발생하는 예외에 대한 알림을 받기를 원하기 때문입니다. 프로덕션 용도로는 `raiseExceptions` 를 `False` 로 설정하는 것이 좋습니다.

---

6 임의의 객체를 메시지로 사용하기

앞의 절과 예제에서 이벤트를 로깅할 때 전달되는 메시지는 문자열이라고 가정했습니다. 그러나 이것만 가능한 것은 아닙니다. 임의의 객체를 메시지로 전달할 수 있으며, 로깅 시스템이 이를 문자열 표현으로 변환해야 할 때 `__str__()` 메서드가 호출됩니다. 사실, 원한다면, 문자열 표현을 계산하는 것을 완전히 피할 수 있습니다 - 예를 들어, `SocketHandler` 는 이벤트를 피클링해서 와이어를 통해 전송하는 방식으로 이벤트를 보냅니다.

7 최적화

메시지 인자의 포매팅은 피할 수 없을 때까지 연기됩니다. 하지만, 로깅 메서드에 전달된 인자를 계산하는 것 또한 비용이 많이들 수 있고, 로거가 이벤트를 버리는 경우에는 수행하지 않고 싶을 수 있습니다. 꼭 해야 할 것을 결정하기 위해서, level 인자를 취하고 해당 수준의 호출이 로거에 의해 이벤트가 생성되면 참을 반환하는 `isEnabledFor()` 메서드를 호출 할 수 있습니다. 다음과 같은 코드를 작성할 수 있습니다:

```
if logger.isEnabledFor(logging.DEBUG):
    logger.debug('Message with %s, %s', expensive_func1(),
                                        expensive_func2())
```

그러면, 로거의 수준이 `DEBUG` 보다 높게 설정된 경우, `expensive_func1()` 과 `expensive_func2()` 호출은 절대 일어나지 않습니다.

참고: 때에 따라, `isEnabledFor()` 자체가 원하는 것보다 더 비쌀 수 있습니다(예를 들어, 명시적 수준이 로거 계층의 상단에서만 설정된 깊이 중첩된 로거의 경우). 그러한 경우(또는 꽉 짜인 루프 내에서 메서드를 호출하지 않으려는 경우) 지역이나 인스턴스 변수에 `isEnabledFor()` 의 호출 결과를 캐시하고, 매번 메서

드를 호출하는 대신 그 값을 사용할 수 있습니다. 이러한 캐시 된 값은 응용 프로그램이 실행되는 동안 로깅 구성이 동적으로 변경될 때(그리 흔한 경우는 아닙니다)만 재계산될 필요가 있습니다.

로깅 정보를 수집하는 방법을 보다 정밀하게 제어해야 하는 특정 응용 프로그램에 대해 수행할 수 있는 다른 최적화가 있습니다. 다음은 필요하지 않은 로깅 처리를 피하고자 수행할 수 있는 작업 목록입니다:

수집하고 싶지 않은 것	수집하는 것을 피하는 방법
호출이 이루어진 위치에 관한 정보.	`logging._srcfile` 을 `None` 으로 설정하십시오. 이렇게 하면 `sys._getframe()` 호출을 피할 수 있는데, PyPy(`sys._getframe()` 을 사용하는 코드의 속도를 높일 수 없습니다)와 같은 환경에서 코드 속도를 높이는 데 도움이 됩니다.
스레딩 정보.	`logging.logThreads` 를 `0` 으로 설정하십시오.
프로세스 정보.	`logging.logProcesses` 를 `0` 으로 설정하십시오.

또한, 코어 logging 모듈에는 기본 처리기만 포함됩니다. `logging.handlers` 와 `logging.config` 를 임포트 하지 않으면 메모리를 차지하지 않습니다.

더 보기:

모듈 `logging` logging 모듈에 관한 API 레퍼런스.

모듈 `logging.config` logging 모듈용 구성 API.

모듈 `logging.handlers` logging 모듈에 포함된 유용한 처리기.

로깅 요리책

색인

Non-alphabetical

R