
Logging

Release 3.11.14

Guido van Rossum and the Python development team

outubro 15, 2025

Python Software Foundation
Email: docs@python.org

Sumário

1	Tutorial Básico de Logging	2
1.1	Quando usar logging	2
1.2	Um exemplo simples	3
1.3	Logging em um arquivo	3
1.4	Logging de variáveis dinâmicas	4
1.5	Alterar o formato das mensagens exibidas	5
1.6	Exibindo data/hora em mensagens	5
1.7	Próximos Passos	6
2	Tutorial Avançado do Logging	6
2.1	Fluxo de Logging	7
2.2	Registradores	8
2.3	Manipuladores	9
2.4	Formatadores	9
2.5	Configurando Logging	10
2.6	O que acontece se nenhuma configuração é fornecida	13
2.7	Configurando logging para uma biblioteca	13
3	Níveis de Logging	14
3.1	Níveis personalizados	15
4	Handlers úteis	15
5	Exceções levantadas durante logging	16
6	Usando objetos arbitrários como mensagens	16
7	Optimização	17
	Índice	18

Autor

Vinay Sajip <vinay_sajip@red-dove.com>

1 Tutorial Básico de Logging

Logging é uma maneira de rastrear eventos que acontecem quando algum software é executado. O desenvolvedor de software adiciona chamadas de logging no código para indicar que determinado evento ocorreu. Um evento é definido por uma mensagem descritiva que pode opcionalmente conter o valor de uma variável (ex.: um dado que é potencialmente diferente pra cada ocorrência do evento). Eventos também possuem importâncias atribuídas pelo desenvolvedor; esta importância pode ser chamada de *nível* ou *severidade*.

1.1 Quando usar logging

Você pode instanciar loggers com `logger = getLogger(__name__)` e, em seguida, chamar os métodos `debug()`, `info()`, `warning()`, `error()` e `critical()`. Para determinar a necessidade ou não de usar logging, e escolher o método adequado, consulte a tabela abaixo. Ela indica, para cada conjunto de tarefas comuns, a melhor ferramenta a ser usada.

Tarefa que você deseja executar	A melhor ferramenta para a tarefa
Exibir uma mensagem na saída do console para uso ordinário de um script de linha de comando (CLI) ou programa.	<code>print()</code>
Expor eventos que possam ocorrer durante a operação normal de um programa (por exemplo, para monitorar o estado atual de execução ou poder investigar falhas)	Utilize o método <code>info()</code> de um logger (ou o método <code>debug()</code> , no caso de mensagens mais detalhadas para diagnóstico e investigação de erros)
Emitir um aviso sobre um evento que ocorre em um tempo de execução específico	Utilize a função <code>warnings.warn()</code> da biblioteca padrão se o problema for evitável, e caso a aplicação deva ser modificada pelo usuário para eliminar o alerta. Utilize o método <code>warning()</code> de um logger, caso não haja nada que o usuário possa fazer na aplicação, mas o evento ainda deva ser registrado
Expor um erro sobre um evento em um tempo de execução específico	Levantando uma exceção
Expor a supressão de um erro sem levantar uma exceção (por exemplo, ao utilizar um tratador de erro em um processo de longa execução)	Um método <code>error()</code> , <code>exception()</code> ou <code>critical()</code> do logger, conforme apropriado para o erro específico e domínio da aplicação

Os métodos de logger são nomeados de acordo com o nível ou severidade dos eventos que eles são usados para rastrear. Os níveis padrão e sua aplicabilidade são descritos abaixo (em ordem crescente de severidade):

Nível	Quando é usado
DEBUG	Informação detalhada, tipicamente utilizáveis apenas para diagnosticar problemas.
INFO	Confirmação de que as coisas estão funcionando como esperado.
WARNING	Uma indicação que algo inesperado aconteceu, ou um indicativo de que poderá haver algum problema em um futuro próximo (por exemplo: 'pouco espaço em disco'). O software está ainda funcionando como esperado.
ERROR	Por conta de um problema mais grave, o software não conseguiu executar alguma função.
CRITICAL	Um erro grave, indicando que o programa pode não conseguir continuar rodando.

O nível padrão é `WARNING`, que significa que só eventos deste nível e acima serão rastreados, a não ser que o pacote logging esteja configurado para fazer de outra forma.

Eventos que são rastreados podem ser tratados de diferentes formas. O jeito mais simples de lidar com eventos rastreados é exibi-los no console. Outra maneira comum é gravá-los em um arquivo de disco.

1.2 Um exemplo simples

Um exemplo bastante simples é:

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

Se você colocar essas linhas no script e executá-lo, você verá:

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

no console. A mensagem INFO não aparece porque o nível padrão de logging é WARNING. A mensagem exibida inclui a indicação do nível e a descrição do evento fornecido na chamada de logging. Ou seja, 'Cuidado!'. A saída pode ser formatada de maneira bastante flexível, se necessário; as opções de formatação também serão explicadas mais tarde.

Observe que neste exemplo, usamos funções diretamente no módulo logging, como logging.debug, ao invés de criar um logger e chamar funções nele. Essas funções operam no logger raiz, mas podem ser úteis, pois chamarão basicConfig() para você se ele ainda não tiver sido chamado, como neste exemplo. Em programas maiores, você geralmente desejará controlar explicitamente a configuração de logging, no entanto - então, por esse motivo, assim como por outros, é melhor criar loggers e chamar seus métodos.

1.3 Logging em um arquivo

Uma situação muito comum é a de registrar eventos de log em um arquivo, então vamos dar uma olhada nisso a seguir. Teste o código a seguir em um interpretador Python recém-iniciado. Não continue a partir da sessão usada para o código anterior.

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

Alterado na versão 3.9: O argumento de codificação *encoding* foi adicionado. Em versões anteriores do Python, a codificação utilizada por padrão seria definida em open(). Embora não seja mostrado no exemplo acima, um argumento de erro pode ser utilizado, o que determina como a codificação de erros são tratadas. Para saber mais sobre essas possibilidades, consulte a documentação de open().

E agora se nós abrirmos o arquivo e olharmos o que temos, deveremos encontrar essas mensagens de log:

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

Este exemplo também mostra como você pode configurar o nível do logging, que funcionará como um limite para o rastreamento. Neste caso, como definimos o nível limite como DEBUG, todas as mensagens foram exibidas.

Você também pode definir o nível de logging a partir de um parâmetro pela linha de comando (CLI):

```
--log=INFO
```

e se você quiser obter o valor do parâmetro `--log` em algum ponto do código, você pode usar:

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

para obter o valor que você passará para a `basicConfig()` pelo argumento de nível *level*. Isso pode ser útil para verificar erros introduzidos pelo usuário, como no exemplo a seguir:

```
# 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, ...)
```

A chamada para `basicConfig()` deve vir *antes* de qualquer chamada para métodos de um logger, como `debug()`, `info()`, etc. Caso contrário, o logging deste evento pode não ser tratado da forma desejada.

Se você executar o script acima diversas vezes, as mensagens das sucessivas execuções serão acrescentadas ao arquivo *exemplo.log*. Se você quiser que cada execução seja iniciada novamente, não guardando as mensagens das execuções anteriores, você pode definir o argumento *filemode*, mudando a chamada no exemplo acima para:

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

A saída será a mesma de antes, mas o arquivo de log não será mais incrementado. Desta forma, as mensagens de execuções anteriores serão perdidas.

1.4 Logging de variáveis dinâmicas

Para logar o dado de uma variável, use o formato string para a mensagem descritiva do evento e adicione a variável como argumento. Exemplo:

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

exibirá:

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

Como você pode ver, para combinar uma variável de dados na mensagem descritiva do evento usamos o velho operador `%`, formatador de strings. Isto é usado para garantir compatibilidade com as versões anteriores: os pacotes logging mais recentes contam com opções de formatação como `str.format()` e `string.Template`. Mas explorar estes métodos está fora do escopo deste tutorial. Veja *formatting-styles* para mais informações.

1.5 Alterar o formato das mensagens exibidas

Para mudar o formato usado para exibir mensagens, você precisa especificar o formato que quer usar:

```
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')
```

que vai exibir:

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

Note que a palavra ‘root’ que apareceu nos exemplos anteriores desapareceu. Para todas as configurações que possam aparecer na formatação de strings, você pode consultar a documentação `logrecord-attributes`, mas para uso simples, você só precisa do `levelname` (severidade), `message` (descrição do evento, incluindo a variável com dados) e talvez exibir quando o evento ocorreu. Isto está descrito na próxima seção.

1.6 Exibindo data/hora em mensagens

Para exibir a data e hora de um evento, você pode colocar `%(asctime)s` na sua string de formato:

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

que deve exibir algo assim:

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

O formato padrão para data/hora (mostrado abaixo) é a ISO8601 ou a **RFC 3339**. Se você precisar de mais controle sobre a formatação de data/hora, defina o argumento `datefmt` na função `basicConfig`, como neste exemplo:

```
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.')
```

que deve exibir algo assim:

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

O formato do argumento `datefmt` é o mesmo suportado por `time.strftime()`.

1.7 Próximos Passos

Concluímos aqui o tutorial básico. Isto deve ser o bastante para você começar a trabalhar com logging. Existe muito mais que o pacote de logging pode oferecer, mas para ter o melhor disto, você precisará investir um pouco mais do seu tempo lendo as próximas seções. Se você está pronto para isso, pegue sua bebida favorita e continue.

If your logging needs are simple, then use the above examples to incorporate logging into your own scripts, and if you run into problems or don't understand something, please post a question on the comp.lang.python Usenet group (available at <https://groups.google.com/g/comp.lang.python>) and you should receive help before too long.

Ainda por aqui? Você pode continuar lendo as seções seguintes, que tem um tutorial mais avançado que o básico acima. Depois disso, você pode dar uma olhada no logging-cookbook.

2 Tutorial Avançado do Logging

A biblioteca de logging tem uma abordagem modular e oferece algumas categorias de componentes: loggers, manipuladores, filtros, e formataadores.

- Loggers expõem a interface que o código da aplicação usa diretamente.
- Handlers enviam os registros do evento (criados por loggers) aos destinos apropriados.
- Filters fornecem uma facilidade granular para determinar quais registros de eventos enviar à saída.
- Formatters especificam o layout dos registros de eventos na saída final.

Um evento, quando registrado, passa por loggers, manipuladores, filtros e formataadores através de uma instância de `LogRecord`

Para executar uma ação de logging, é necessário criar instâncias da classe `Logger` (também chamadas de *loggers*), e invocar seus métodos. Cada instância tem um nome, e elas são conceitualmente organizadas em uma hierarquia de espaço de nomes utilizando pontos como separadores. Por exemplo, um logger nomeado como 'leitor' é o pai do logger 'leitor.texto', 'leitor.html' e 'leitor.pdf'. Você pode nomear o logger do jeito que preferir, e indicar o trecho da aplicação que origina a mensagem de log.

Uma boa prática ao nomear loggers é criar, em cada módulo que usa logging, um logger definido no mesmo nível do módulo. Como no exemplo abaixo:

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

Isso significa que os nomes dos loggers acompanham a hierarquia de pacotes e módulos, tornando intuitivo saber de onde os eventos foram registrados apenas olhando o nome do logger.

O início da hierarquia de loggers é chamado de logger raiz. É ele que é usado pelas funções `debug()`, `info()`, `warning()`, `error()` e `critical()`, que apenas chamam o método de mesmo nome dentro do logger raiz. Essas funções, e seus métodos correspondentes, têm a mesma assinatura (mesmos parâmetros aceitos). Na saída do log, o nome do logger raiz aparece como 'root'.

É possível registrar mensagens de log em diferentes destinos. O pacote já traz suporte para enviar logs para arquivos, URLs via HTTP GET/POST, e-mails via SMTP, soquetes genéricos, filas ou mecanismos de log específicos do sistema operacional, como o syslog no Unix/Linux ou o Event Log do Windows NT. Esses destinos são gerenciados por classes chamadas manipuladores. Se precisar de um destino especial que não seja atendido pelos manipuladores embutidos, você poderá criar sua própria classe manipuladora para destinar corretamente o log.

Por padrão, nenhum destino é definido para nenhuma mensagem de registro. É possível especificar um destino (como console, ou arquivo) usando `basicConfig()`, como nos exemplos do tutorial. Se você chamar as funções `debug()`, `info()`, `warning()`, `error()` e `critical()`, elas verificarão se o destino está definido; e, caso não esteja,

elas definirão o console como o destino (`sys.stderr`) e definirão o formato utilizando um formato padrão. Só então, enviarão a mensagem ao logger raiz, para que seja registrada na saída.

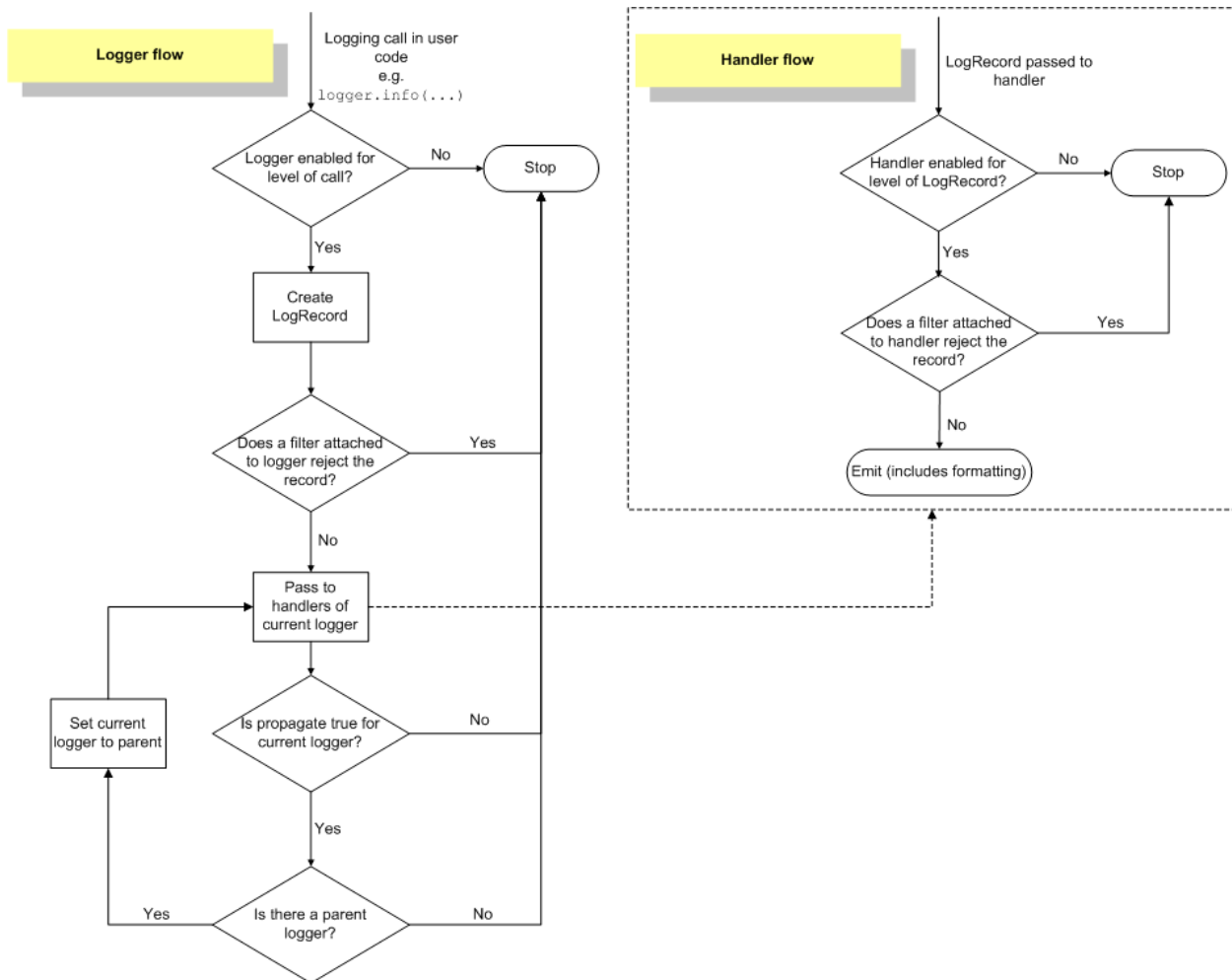
O formato padrão definido por `basicConfig()` para mensagens é:

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

Você pode alterar isso passando uma string de formato para `basicConfig()`, através do argumento nomeado *format*. Para sobre como uma string de formato é construída, consulte `formatter-objects`.

2.1 Fluxo de Logging

O fluxo dos eventos de log em loggers e manipuladores é ilustrado no diagrama a seguir.



2.2 Registradores

Objetos `Logger` têm três funções principais. Primeiro, eles expõem vários métodos para que o código da aplicação possa registrar mensagens durante a execução. Segundo, eles decidem quais mensagens de log devem ser processadas, com base no nível de severidade (é o filtro padrão) ou em objetos filtro. Terceiro, eles encaminham as mensagens de log relevantes para todos os manipuladores de log interessados.

Os métodos mais utilizados em objetos logger se enquadram em duas categorias: configuração e envio de mensagem.

Esses são os métodos de configuração mais comuns:

- `Logger.setLevel()` especifica o nível mínimo de severidade que um logger vai lidar. O nível `DEBUG` é o mais baixo, e o `CRITICAL` é o mais alto entre os níveis padrão. Por exemplo, se o nível do logger for `INFO`, ele vai processar apenas mensagens de nível `INFO`, `WARNING`, `ERROR` e `CRITICAL`, e vai ignorar mensagens `DEBUG`.
- `Logger.addHandler()` e `Logger.removeHandler()` adicionam e removem objetos manipuladores do objeto logger. Manipuladores serão aprofundados em [Manipuladores](#).
- `Logger.addFilter()` e `Logger.removeFilter()` adicionam e removem objetos filtros do objeto logger. Filtros serão aprofundados em [filter](#).

Você não precisa sempre invocar esses métodos em cada logger que criar. Consulte os últimos dois parágrafos nessa mesma seção.

Com o objeto logger configurado, os seguintes métodos criam mensagens de log:

- `Logger.debug()`, `Logger.info()`, `Logger.warning()`, `Logger.error()` e `Logger.critical()` criam logs com uma mensagem em um nível correspondente ao respectivo nome do método. A mensagem é uma string de formato, que pode conter a sintaxe padrão de substituições como `%s`, `%d` e `%f`. O restante dos argumentos são uma lista de objetos que correspondem aos campos de substituição da mensagem. No caso do parâmetro `**kwargs`, os métodos de logging cuidam apenas da chave `exc_info`. É essa chave que eles usam para decidir se devem ou não registrar os detalhes de uma exceção.
- `Logger.exception()` cria uma mensagem de log parecida com `Logger.error()`. A diferença é que, além da mensagem, `Logger.exception()` também registra o stack trace (situação da pilha de execução). Chame este método apenas a partir de um manipulador de exceção.
- `Logger.log()` recebe o nível de log como um argumento explícito. Isso torna o registro de mensagens um pouco mais verboso do que usar os métodos prontos para cada nível, com os métodos listados anteriormente, mas é uma forma de registrar mensagens em níveis de log personalizados.

`getLogger()` retorna uma referência para uma instância de um logger com o nome especificado, caso informado, ou para o logger raiz (caso nenhum nome seja passado). Os nomes seguem uma estrutura hierárquica separada por pontos. Chamadas repetidas a `getLogger()` com o mesmo nome retornam sempre uma referência para a mesma instância do objeto logger. Loggers que aparecem mais embaixo na hierarquia são filhos dos loggers que estão mais acima nessa lista. Por exemplo, dado um logger com nome `foo`, loggers com os nomes `foo.bar`, `foo.bar.baz` e `foo.bam` serão todos descendentes de `foo`.

Loggers têm o conceito de nível efetivo. Se um logger não tiver um nível definido explicitamente, ele herdará o nível do seu logger pai. Caso o pai também não tenha um nível definido, o processo continua subindo na hierarquia, examinando cada ancestral, até encontrar um logger com nível configurado. O logger raiz sempre terá um nível explícito configurado (por padrão, `WARNING`). Ao decidir se um evento será processado, o logger usa o seu nível efetivo para determinar se o evento deve ser repassado para os manipuladores responsáveis.

Loggers filhos propagam suas mensagens para os manipuladores associados aos seus loggers ancestrais. Por causa disso, não é necessário definir e configurar manipuladores para todos os loggers usados em uma aplicação. Basta configurar os manipuladores em um logger de nível superior, e criar loggers filhos conforme a necessidade. No entanto, é possível desativar essa propagação definindo o atributo *propagate* do logger como `False`.

2.3 Manipuladores

Objetos `Handler` são responsáveis por enviar as mensagens de log apropriadas (com base na severidade de cada mensagem) para o destino especificado do manipulador. Objetos `Logger` podem adicionar um ou mais manipuladores a si mesmos usando o método `addHandler()`. Em um cenário hipotético, uma aplicação pode querer “enviar todas as mensagens de log para um arquivo”, “enviar apenas mensagens de erro ou mais graves para o *stdout*” e “enviar apenas mensagens críticas para um endereço de e-mail”. Esse cenário exigiria três manipuladores diferentes, cada um responsável por encaminhar mensagens de uma determinada gravidade para um destino específico.

A biblioteca padrão possui alguns tipos de manipuladores (veja *Handlers úteis*); os tutoriais usam principalmente `StreamHandler` e `FileHandler` em seus exemplos.

Existem pouquíssimos métodos em um manipulador com os quais desenvolvedores de aplicações precisam se preocupar. Para quem está usando manipuladores já incluídos na biblioteca (ou seja, sem criar manipuladores personalizados), os únicos métodos realmente relevantes são os métodos de configuração a seguir:

- O método `setLevel()`, assim como acontece nos objetos `logger`, define o menor nível de severidade das mensagens que serão encaminhadas para o destino adequado. Mas por que existem dois métodos `setLevel()`? Isso acontece porque o nível configurado no `logger` define quais mensagens ele vai repassar para seus manipuladores. Já o nível configurado em cada manipulador define, entre as mensagens recebidas, quais realmente serão enviadas adiante.
- `setFormatter()` define um objeto `Formatador` para que o manipulador utilize.
- `addFilter()` e `removeFilter()`, respectivamente, adicionam e removem objetos filtros nos manipuladores.

O código da aplicação não deve instanciar nem usar diretamente objetos da classe `Handler`. Na verdade, `Handler` é uma classe base que define a interface que todos os manipuladores devem implementar e estabelece um comportamento padrão que pode ser aproveitado (ou sobrescrito) pelas classes filhas.

2.4 Formataadores

Objetos `Formatadores` definem a ordem, a estrutura e o conteúdo final da mensagem de log. Diferentemente da classe base `logging.Handler`, o código da aplicação pode instanciar classes de formataadores — embora também seja possível criar subclasses, caso sua aplicação precise de um comportamento específico. O construtor aceita três argumentos opcionais: uma string de formato da mensagem, uma string de formatação da data e um indicador de estilo.

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

Se nenhuma string de formato for fornecida na mensagem, o padrão é usar a mensagem bruta. Se nenhuma string de formato de data for definida, o formato de data padrão será:

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

com os milissegundos acrescentados ao final. O parâmetro `style` pode ser `'%'`, `'{'` ou `'$'`. Se nenhum deles for especificado, o padrão utilizado será `'%'`.

Se o parâmetro `style` for `'%'`, a string de formato da mensagem substituirá a string usando o estilo `% (<chave de dicionário> s)`; as chaves possíveis estão documentadas em `logrecord-attributes`. Se o estilo for `'{'`, a string de formato é presumida como compatível com `str.format()` (usando argumentos nomeados). Já se o estilo for `'$'`, a string de formato deverá seguir o que é esperado por `string.Template.substitute()`.

Alterado na versão 3.2: Adicionado o parâmetro `style`.

A string de formato a seguir vai escrever o tempo em um formato legível para humanos, a severidade da mensagem, e o conteúdo da mensagem, nessa ordem:

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

Formatadores usam uma função configurável pelo usuário para converter o horário de criação de um registro em uma tupla. Por padrão, é utilizada a função `time.localtime()`; Se você quiser alterar isso apenas para uma instância específica de formatador, basta definir o atributo `converter` dessa instância para uma função com a mesma assinatura de `time.localtime()` ou `time.gmtime()`. Já se a ideia for alterar o comportamento para todos os formatadores (por exemplo, para exibir todos os horários de log em GMT), defina o atributo `converter` diretamente na classe Formatadora (usando, por exemplo, `time.gmtime` para exibição em GMT).

2.5 Configurando Logging

Programadores podem configurar logging de três formas:

1. Criando loggers, manipuladores e formatadores de forma explícita, usando código Python que invoca os métodos de configuração mencionados acima.
2. Criando um arquivo de configuração de logging e lendo-o usando a função `fileConfig()`.
3. Criando um dicionário com as informações de configuração e passando-o para a função `dictConfig()`.

Para a documentação de referência sobre as duas últimas opções, consulte `logging-config-api`. O exemplo a seguir configura um logger bem simples, um manipulador de console e um formatador simples utilizando código Python:

```
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')
```

A execução deste módulo pela linha de comando produz a seguinte saída:

```
$ 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
```

O módulo Python a seguir cria um logger, um manipulador e um formatador quase idênticos aos do exemplo mostrado acima, sendo a única diferença os nomes dos objetos:

```
import logging
import logging.config

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')
```

Aqui está o arquivo logging.conf:

```
[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
```

A saída é quase idêntica à do exemplo não baseado em arquivo de configuração:

```
$ 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
```

A abordagem de usar um arquivo de configuração oferece algumas vantagens em relação ao uso direto de código Python. Ela permite separar a configuração do código, facilitando a manutenção, e também possibilita que pessoas sem conheci-

mentos de programação modifiquem facilmente as propriedades de logging apenas editando o arquivo de configuração.

Aviso: A função `fileConfig()` possui um parâmetro chamado `disable_existing_loggers`, que por padrão é definido como `True` por questões de retrocompatibilidade. No entanto, esse comportamento pode não ser o desejado, pois todos os loggers que não o raiz que já existiam antes da chamada de `fileConfig()` serão desativados, a menos que eles (ou algum de seus ancestrais) sejam explicitamente nomeados na configuração. Caso queira manter os loggers preexistentes ativos, basta definir esse parâmetro como `False`. Para maiores detalhes, consulte a documentação oficial de referência.

O dicionário passado para `dictConfig()` também pode especificar um valor booleano com a chave `disable_existing_loggers`. Se essa chave não for definida explicitamente no dicionário, o valor padrão será interpretado como `True`. Isso resulta no mesmo comportamento de desativação de loggers descrito anteriormente, o que pode não ser o desejado. Nesse caso, basta informar a chave no dicionário com o valor `False`.

Observe que os nomes de classe referenciados em arquivos de configuração precisam ser relativos ao módulo logging, ou valores absolutos que possam ser resolvidos pelos mecanismos normais de importação do Python. Por exemplo, você pode usar `WatchedFileHandler` (relativo ao módulo logging) ou `mypackage.mymodule.MyHandler` (para uma classe definida no pacote `mypackage` e no módulo `mymodule`, desde que `mypackage` esteja disponível no caminho de importação do Python).

A partir do Python 3.2, foi introduzida uma nova forma de configurar o logging, utilizando dicionários para armazenar as informações de configuração. Essa abordagem oferece um conjunto ampliado de funcionalidades em relação ao método baseado em arquivos de configuração mencionado anteriormente, sendo, portanto, a forma recomendada de configuração para novas aplicações e implantações. Como a configuração é mantida em um dicionário Python, há maior flexibilidade sobre as opções de como esse dicionário pode ser construído. É possível, por exemplo, utilizar um arquivo em formato JSON ou YAML, construir o dicionário diretamente em código Python, recebê-lo em forma serializada com pickle por meio de um soquete, ou ainda empregar qualquer outro método que faça sentido para a aplicação em questão.

Aqui está um exemplo da mesma configuração acima, em formato YAML, para a nova abordagem baseada em dicionário:

```
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]
```

Para mais informações sobre logging com dicionário, consulte [logging-config-api](#).

2.6 O que acontece se nenhuma configuração é fornecida

Se nenhuma configuração de logging for fornecida, pode ocorrer uma situação em que um evento de logging precise ser registrado, mas nenhum manipulador esteja disponível para processar e exibir esse evento.

O evento é exibido utilizando um “manipulador de último recurso”, armazenado em `lastResort`. Esse manipulador interno não está associado a nenhum logger e funciona como um `StreamHandler`, enviando a mensagem de descrição do evento para o valor atual de `sys.stderr` (espeitando, portanto, quaisquer redirecionamentos que estejam em vigor). Nenhuma formatação é aplicada à mensagem — apenas a descrição bruta do evento é impressa. O nível desse manipulador é definido como `WARNING`, de forma que todos os eventos nesse nível ou em níveis mais severos serão exibidos.

Alterado na versão 3.2: Para versões do Python anteriores à 3.2, o comportamento é o seguinte:

- Se `raiseExceptions` for `False` (modo em produção), o evento será silenciosamente descartado.
- Se `raiseExceptions` for definido como `True` (modo de desenvolvimento), a mensagem ‘No handlers could be found for logger X.Y.Z’ será exibida uma vez.

Para obter o comportamento pré-versão-3.2, `lastResort` pode ser definido como `None`.

2.7 Configurando logging para uma biblioteca

Ao desenvolver uma biblioteca que usa logging, é importante documentar como a biblioteca usa o recurso - por exemplo, os nomes dos loggers usados. Também é necessário levar em consideração a configuração de logging. Caso a aplicação que utiliza a biblioteca não faça uso de logging e o código da biblioteca execute chamadas de logging, então os eventos de severidade `WARNING` e níveis mais severos serão impressos em `sys.stderr` (conforme descrito na seção anterior). Isso garante que mensagens importantes não sejam perdidas, como comportamento padrão, mesmo que a aplicação principal não configure explicitamente o sistema de logging.

Se, por algum motivo, você *não* quiser que essas mensagens sejam exibidas na ausência de qualquer configuração de logging, pode anexar um manipulador que não faz nada ao logger de nível superior da sua biblioteca. Dessa forma, evita-se que a mensagem seja impressa, já que sempre haverá um manipulador associado aos eventos da biblioteca — apenas não produzirá saída. Caso o usuário da biblioteca configure o logging para uso na aplicação, essa configuração provavelmente incluirá alguns manipuladores adicionais e, se os níveis forem configurados adequadamente, as chamadas de logging feitas no código da biblioteca enviarão a saída para esses manipuladores, funcionando normalmente.

Um manipulador-que-não-faz-nada já está incluso no pacote `NullHandler` (desde Python 3.1). Uma instância desse manipulador poderia ser adicionado ao logger de nível superior do espaço de nomes de logging usado pela biblioteca (se você quiser evitar que os eventos registrados da biblioteca sejam enviados para `sys.stderr`, na ausência de configuração de logging). Se todo registro em log de uma biblioteca *foo* for feito usando loggers com nomes correspondente ‘foo.x’, ‘foo.x.y’ etc., então o código:

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

deve resultar no efeito desejado. Se uma organização produzir várias bibliotecas, o nome do logger especificado poderá ser ‘orgname.foo’ em vez de apenas ‘foo’.

Nota: É altamente recomendável que você *não registre logs no logger raiz* do seu biblioteca. Em vez disso, use um logger com um nome exclusivo e facilmente identificável, como o `__name__` do nível superior do seu pacote/módulo da biblioteca. O registro em log no logger raiz dificultará, ou mesmo tornará impossível que o desenvolvedor da aplicação personalize o manipulador e a verbosidade de logging.

Nota: É altamente recomendável que você *não adicione outros manipuladores além de `NullHandler` nos loggers da sua biblioteca*. Isso porque a configuração de manipuladores é responsabilidade do desenvolvedor da aplicação que

utilizará ela. O desenvolvedor sabe o público-alvo da aplicação final, e, portanto, sabe também quais são os manipuladores mais adequados para o caso. Se você adicionar manipuladores ‘ocultos’, poderá interferir na criação de testes unitários e na entrega de logs úteis.

3 Níveis de Logging

Os valores numéricos dos níveis de logging estão listados na tabela abaixo. Eles são principalmente de interesse se você quiser definir seus próprios níveis, e precisa deles para definir seus valores específicos relativos aos níveis predefinidos. Se você define um nível com o mesmo valor numérico, ele sobrescreve o valor predefinido; o nome predefinido é perdido.

Nível	Valor numérico
CRITICAL	50
ERROR	40
WARNING	30
INFO	20
DEBUG	10
NOTSET	0

Os níveis também podem ser associados aos registradores, sendo definidos pelo desenvolvedor ou por meio do carregamento de uma configuração salva. Quando um método de logging é invocado em um logger, o logger compara o seu próprio nível com o nível associado à chamada do método. Se o nível do logger for maior do que o da chamada do método, nenhuma mensagem será gerada. Esse é o mecanismo básico que controla a verbosidade da saída de logging.

As mensagens de logging são representadas como instâncias da classe `LogRecord`. Quando um logger decide registrar um evento, uma instância de `LogRecord` é criada a partir da mensagem de logging.

As mensagens de logging passam por um mecanismo de despacho que utiliza *manipuladores*, que são instâncias de subclasses da classe `Handler`. Manipuladores são responsáveis por assegurar que uma mensagem registrada em log (na forma de `LogRecord`) seja encaminhada para um determinado destino (ou conjunto de destinos) que serão úteis para o público-alvo daquela mensagem (como usuários finais, equipe de suporte, administradores do sistema e desenvolvedores). Manipuladores transmitem `LogRecord` para destinos específicos. Cada logger pode ter zero, um ou mais manipuladores associados à ele (através do método `addHandler()` de `Logger`). Além disso, não são apenas os manipuladores diretamente associados a um logger que processam a mensagem. Por padrão, os manipuladores de todos os loggers ancestrais também participam do despacho. Essa propagação só é interrompida se o atributo *propagate* do logger estiver definido como `False`.

Assim como os loggers, os manipuladores também podem ter níveis associados. O nível de um manipulador funciona como um filtro, do mesmo modo que o nível de um logger. Quando um manipulador decide processar um evento, ele utiliza o método `emit()` para enviar a mensagem ao seu destino. Na maioria dos casos, subclasses personalizadas de `Handler` precisarão substituir esse método `emit()`.

3.1 Níveis personalizados

Criar seus próprios níveis é possível, mas não deve ser necessário, pois os níveis existentes foram escolhidos com base na experiência prática. No entanto, se for estritamente necessário, tome cuidado para fazer. E se você está *desenvolvendo uma biblioteca*, não é recomendado que defina níveis personalizados. Isso porque, se vários autores de biblioteca definirem seus próprios níveis personalizados, as saídas de logging das bibliotecas em conjunto serão orquestradas com maior dificuldade pelo desenvolvedor, pois um determinado valor numérico poderá significar coisas diferentes para diferentes bibliotecas.

4 Handlers úteis

Em adição à classe base `Handler`, muitas subclasses úteis são fornecidas:

1. `StreamHandler`, quando instanciadas, enviam mensagens para streams (objeto arquivo ou similar).
2. `FileHandler`, quando instanciadas, enviam mensagens para arquivos locais de disco.
3. `BaseRotatingHandler` é a classe base para objetos handler que rotacionam os arquivos de logging em um determinado ponto. Não é feita para ser instanciada diretamente. Em vez disso, use `RotatingFileHandler` ou `TimedRotatingFileHandler`.
4. `RotatingFileHandler`, quando instanciadas, enviam mensagens para arquivos locais de disco. Permite definir o tamanho máximo do arquivo de log e configurar rotação.
5. `TimedRotatingFileHandler`, quando instanciadas, enviam mensagens para arquivos locais de disco, rotacionando o arquivo de log em certos intervalos de tempo.
6. `SocketHandler`, quando instanciadas, enviam mensagens para soquetes TCP/IP. Desde a versão 3.4, os soquetes de domínio Unix também são suportados.
7. `DatagramHandler`, quando instanciadas, enviam mensagens para soquetes UDP. Desde a versão 3.4, os soquetes de domínio Unix também são suportados.
8. `SMTPHandler`, quando instanciadas, enviam mensagens para endereços de e-mail pré-definidos.
9. `SysLogHandler`, quando instanciadas, enviam mensagens para um daemon syslog Unix, normalmente em um computador remoto.
10. `NTEventLogHandler`, quando instanciadas, enviam mensagens para um registro de log do Windows NT/2000/XP.
11. `MemoryHandler`, quando instanciadas, enviam mensagens para um buffer na memória, que é liberado sempre que alguns critérios específicos são atendidos.
12. `HTTPHandler`, quando instanciadas, enviam mensagens para um servidor HTTP, utilizando os métodos GET ou POST.
13. `WatchedFileHandler`, quando instanciadas, observam o arquivo que estão fazendo logging. Se o arquivo for alterado, ele será fechado e reaberto com o mesmo nome. Este manipulador só é funcional em sistemas Unix ou similares; o Windows não oferece suporte ao mecanismo subjacente utilizado.
14. `QueueHandler`, quando instanciadas, enviam mensagem para uma fila, que pode ser implementada com o módulo `queue` ou com o módulo `multiprocessing`.
15. `NullHandler`, quando instanciadas, não fazem nada com as mensagens de erro. Elas são usadas por desenvolvedores da biblioteca que precisam usar o logging, mas querem evitar o aviso 'No handlers could be found for logger XXX', que pode ser exibido quando o logging não é configurado. Veja [Configurando logging para uma biblioteca](#) para mais informações.

Novo na versão 3.1: A classe `NullHandler`.

Novo na versão 3.2: A classe `QueueHandler`.

As subclasses `NullHandler`, `StreamHandler` e `FileHandler` são definidas no pacote base de logging. Os outros manipuladores são definidos no submódulo `logging.handlers`. (Existe também outro submódulo, `logging.config`, para configurar as funcionalidades).

As mensagens de log são formatadas por meio de instâncias da classe `Formatter`. Elas são inicializadas com uma string de formato, que pode ser usada junto com o operador `%` e um dicionário para substituir os valores na mensagem.

Para formatar várias mensagens em um lote, podem ser usadas instâncias de `BufferingFormatter`. Além de aplicar string de formato em cada mensagem do lote, é possível definir strings de formato para o cabeçalho e o rodapé do lote.

Quando não for suficiente fazer a filtragem baseada no nível do logger e/ou no nível de manipulador, instâncias de `Filter` poderão ser adicionadas às instâncias de `Logger` e `Handler` (por meio do método `addFilter()`). Antes de processar uma mensagem, os loggers e manipuladores consultam todos os seus filtros para obter permissão. Se algum filtro retornar um valor falso, a mensagem não é processada.

A funcionalidade básica do `Filter` permite a filtragem por um nome específico de logger. Se este recurso for usado, as mensagens enviadas para o logger com esse nome, ou seus descendentes, serão permitidas pelo filtro. E todas as outras são descartadas.

5 Exceções levantadas durante logging

O pacote de logging foi projetado para inibir exceções que possam ocorrer quando os registros forem feitos em produção. Desta forma, a aplicação que está utilizando logging não encerrará abruptamente caso haja erros nos eventos de logging - devido a problemas como má configuração, rede ou outros.

As exceções `SystemExit` e `KeyboardInterrupt` nunca são inibidas. Outras exceções que possam ocorrer durante o método `emit()` de uma subclasse de `Handler` são passadas para seus respectivos métodos `handleError()`.

A implementação padrão de `handleError()` em `Handler` verifica se uma variável de módulo, `raiseExceptions`, está definida. Caso esteja, um traceback (situação da pilha de execução) é exibido em `sys.stderr`. Se não, a exceção é inibida.

Nota: O valor padrão de `raiseExceptions` é verdadeiro. Isso ocorre porque, durante o desenvolvimento, você normalmente desejará ser notificado sobre qualquer exceção que ocorra. É recomendável que você defina `raiseExceptions` como falso para usos em produção.

6 Usando objetos arbitrários como mensagens

Nas seções e exemplos anteriores, todas as mensagens criadas ao realizar o logging de um evento foram instanciadas como string. No entanto, essa não é a única possibilidade. Você pode passar um objeto arbitrário como mensagem, e seu método `__str__()` será chamado quando o sistema de logging precisar convertê-lo em uma string. Inclusive, é possível evitar totalmente o uso de uma representação em string - por exemplo, a classe `SocketHandler` emite a mensagem enviando pela rede sua respectiva serialização com pickle.

7 Otimização

A formatação dos argumentos das mensagens é adiada até o quanto for possível. No entanto, o processamento dos argumentos para o método de logging pode ser custoso, e talvez você queira evitar fazê-lo, se o logging for descartar o evento. Para decidir o que fazer, você pode chamar o método `isEnabledFor()`, que recebe um nível como argumento e retorna verdadeiro apenas se o evento tiver sido criado pelo logger para esse nível em específico. Desta forma, você pode escrever algo como:

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

desta forma, se o limite do logger for definido acima de `DEBUG`, as chamadas para `func_custosa1` e `func_custosa2` nunca serão feitas.

Nota: Em alguns casos, o próprio `isEnabledFor()` pode ser mais custoso do que você gostaria (por exemplo, para loggers profundamente aninhados, em que um nível explícito só é definido no logger mais alto da hierarquia). Nesses casos (ou se você quiser evitar chamar um método em um laço), é possível cachear o resultado de uma chamada para `isEnabledFor()` em uma variável local ou instância, e usá-lo em vez de chamar o método todas as vezes. Este valor armazenado só precisaria ser recalculado se a configuração do logger fosse alterada dinamicamente, com a aplicação em execução (o que não é comum).

Existem outras otimizações que podem ser feitas para aplicação específicas que precisam de um controle mais preciso sobre quais informações de logging são registradas. Abaixo encontra-se uma lista de coisas que você pode fazer para evitar o processamento desnecessário para loggings que serão descartados:

O que você não quer registrar	Como evitar o registro
Informações sobre o ponto em que as chamadas foram feitas.	Defina <code>logging._srcfile</code> como <code>None</code> . Isso evita chamar <code>sys._getframe()</code> , o que pode acelerar seu código em ambientes como o PyPy (que não consegue acelerar códigos que usam <code>sys._getframe()</code>).
Informações sobre threading.	Defina <code>logging.logThreads</code> como <code>False</code> .
ID do processo atual (<code>os.getpid()</code>)	Defina <code>logging.logProcesses</code> como <code>False</code> .
Nome do processo atual, ao utilizar o módulo <code>multiprocessing</code> para gerenciar múltiplos processamentos.	Defina <code>logging.logMultiprocessing</code> como <code>False</code> .

Observe também que o módulo base de logging inclui apenas o manipulador básico. Se você não importar `logging.handlers` e `logging.config`, eles não ocuparão nenhum espaço da memória.

Ver também:

Módulo `logging`

Referência da API para o módulo de logging.

Módulo `logging.config`

API de configuração para o módulo logging.

Módulo `logging.handlers`

Tratadores úteis incluídos no módulo logging.

Um livro de receitas do logging

Índice

Não alfabético

`__init__()` (método `logging.logging.Formatter`), 9

R

RFC

RFC 3339, 5