Developers get the logs they deserve

Denis Kataev

Tinkoff.ru

About me

- 9 years developing in Python
- sqlalchemy fan

Logs?

Why are they needed?

import logging

1. Logger
2. Handler
3. Filter
4. Formatter
5. LogRecord
6. LoggerAdapter

How it works

We need a logger

```
import logging

logging.info(...)
```

We need a logger

```
import logging

logging.info(...)
```

Writing in root logger is bad practice

Create logger

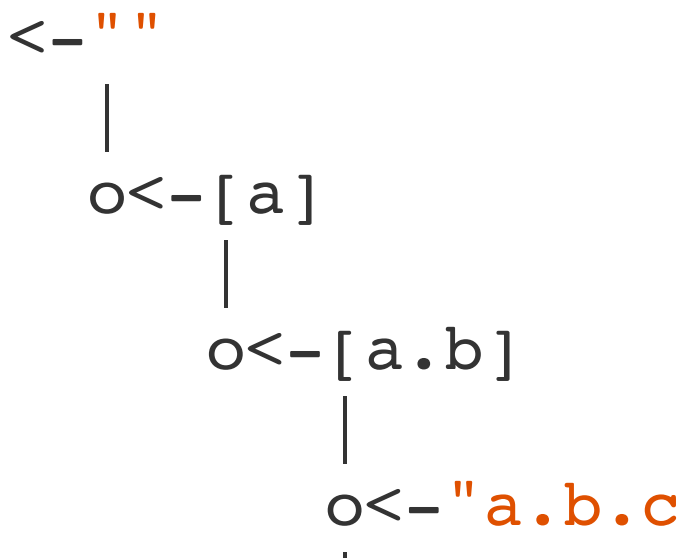
```
logger = logging.getLogger(__name__)

def some_code():
    logger.info(...)
```

Create LogRecord

- log level
- message
- data
- extra

```
logger.info('some value: %s', value)
```

Travel around the logging tree




```
import logging

logging.getLogger('a.b.c')
<Logger a.b.c (WARNING)>
logging.getLogger('a.b.d')
<Logger a.b.d (WARNING)>
```

All roads lead to root logger

root_logger is singleton

- logging.getLogger()
- logging.getLogger("")
- logging.root
- ...

Main idea logging

“ There are events and they pop up

Log level

How to choose

Type of message	**Level**
Exception with trace	.exception
Logical error	.error
Strage or rare cases	.warning
Normal message	.info
Other noise	.debug

Different levels

- record level;
- logger level;
- handler level.

Journey may end prematurely

- If record level less than logger level;
- If logger with propagate=False;
- If the record just didn’t like.

NOTSET level

Default level for all loggers except root logger

```
<--""
 Level WARNING
 |
 o<-[a]
    |
    o<-[a.b]
       |
       o<-"a.b.c"
       |  Level NOTSET so inherits level WARNING
       |
       o<-"a.b.d"
          Level NOTSET so inherits level WARNING
```

Отрываемся от веточки

```
logging.getLogger('a.b').propagate = False
```

```
<--""
 Level WARNING
 |
 o<-[a]
    |
    o  [a.b]
       Level NOTSET so inherits level WARNING
       Propagate OFF
       |
       o<-"a.b.c"
          Level NOTSET so inherits level WARNING
```

Logger filter

Creating your own filtering logic

```
def kek_filter(record: LogRecord):
    return 'kek' not in record.msg

class FilterSubstringText:
    def __init__(self, substring: str):
      self.substring = substring

    def filter(self, record: LogRecord) -> bool:
      return self.substring in record.msg


logger = logging.getLogger('a.b')
logger.addFilter(kek_filter)
logger.addFilter(FilterSubstringText('elapsed'))
```

Flow control

“ There are events and some of them survive

Output from system

Handler

Logger have a handler

```
logger = logging.getLogger('a')
logger.addHandler(logging.StreamHandler())
```

```
<--""
    Level WARNING
    |
    o<--"a"
        Level NOTSET so inherits level WARNING
        Handler Stream <_io.TextIOWrapper
name='<stderr>' mode='w' encoding='UTF-8'>
        |
        o<--[a.b]
            |
            o<--"a.b.c"
                Level NOTSET so inherits level
```

Handler have level too

1. record reached the logger;
2. logger has a handler;
3. handler *ignores* records below its level

Handler also can has filters

1. Record reached the logger (lucky one);
2. and logger have handler;
3. and handler have filter;
4. handler *ignores* messages that did not pass the filter.

Hanlder formatter

Convert LogRecord instance into *text*!

```
class RequestFormatter(logging.Formatter):
    def format(self, record: LogRecord) -> str:
        s = super().format(record)

        value = getattr(record, 'json', None)
        if value:
          s += f'\njson: {pprint.pformat(value)}'
        return s

h = logging.StreamHandler()
h.setFormatter(RequestFormatter())
logging.getLogger('a').addHandler(h)
```

Default formatter

- Specifies strings for formatting a message;
- separately formats event message and time of the event;
- it is usually set in the format %s of the string;
- but there are ways to use a different style (f'strings).

Output control

“ Кручу-верчу сообщения вывожу

How write logs?

Arguments

```
logger.info("Some text %s" % "text")
logger.info(f"Some text {text}")
```

Arguments

```
logger.info("Some text %s" % "text")
logger.info(f"Some text {text}")
```

Arguments

```
logger.info("Some text %s", "text")
```

Errors

```
try:
    some_code()
except Exception as e:
    logger.error(e)

try:
    some_code()
except Exception as e:
    logger.error("Some bad situation %s", e)
```

Errors

```
try:
    some_code()
except Exception as e:
    logger.error(e)

try:
    some_code()
except Exception as e:
    logger.error("Some bad situation %s", e)
```

Errors

```
try:
    some_code()
except Exception:
    logger.exception("Some bad situation")
```

Why is it important?

- If there is a formatting error, the code will not fall;
- formatter will display a traceback for you;
- formatting are lazy and occurs in formatter (suddenly);
- **sentry** groups errors by message template.

```
logger.error(f'Some error {user_id}')
# many simmilar errors in sentry


logger.error('Some error %s', user_id)
# one error in sentry with many examples
```

flake8-logging-format

m-burst commented on 9 Aug 2018 Contributor

I've added some new violations to check:

- Check that logging calls in `except` blocks do not incorporate exceptions into the message (`logger.exception` or is preferred, or `exc_info=True` for level other than ERROR)
- Check for `logger.error(..., exc_info=True)` (`logging.exception` is preferred)
- Check for redundant `logger.exception(..., exc_info=True)`

👍 2

Write correctly

“ Little nuances that mean a lot

What are the handlers we have?

Simple and famous

- StreamHandler;
- FileHandler;
- WatchedFileHandler;
- some RotatingHandler;
- and different network handlers

Cool and standard

- BufferingHandler
- MemoryHandler
- QueueHandler
- ...
- NullHandler

Memory

- append LogRecords to list;
- and sometime sends all records into another handler.

Buffering

- append LogRecords to list;
- and sometime clears the list.

You can write this trigger of certain moment yourself.

Can reduce cpu load

QueueHandler

- Write LogRecord's into queue;
- You create a queue!

QueueListener

- Reading a queue;
- sending records to another handler.

You can use thread safe queue

With this, logging does not block the loop!

NullHandler

King of the handlers

```
No handlers could be found for logger X.Y.Z
```

NullHandler

King of the handlers

If you write a library, you must not add any handlers other than NullHandler!

Sad examples:

- grpc
- timeloop
- and many others

Many and many others

from different libraries

graylog example

```
log_fields = {'event': 'handle_request',
              'client_text': request.get('text')}

with gelf.timelog(log_fields):
        some_code(...)


    ...
    more_code()

    with gelf.timelog(log_fields, **{'a': 'b'}):
        some_other_code(...)
```

graylog example

```
logger.info('Handle event',
            extra={'client_text': request.get('text
some_code(...)

...
more_code()
logger.info('Some success', extra={'a': b})
```

Not just text

- yep write text in console;
- but can send over network something more;
- can have context.

Events context

- x-trace-id
- some pass-through identifier
- ...
- any over information

LogRecord extra

- Dict for any params, what we can use in formatter or handler.
- Since logging is lazy, you need to remember about garbage collection

what the docs thinks about it

we can skip the context in several ways

- thought formatter, filter, etc. (uncool)
- thought LoggerAdapter (better)
- but most useful using setLoggerFactory (override LogRecord creating)

LoggerAdapter

Wrapping logger and add dict to extra each LogRecord

```
logger = logging.getLogger(__name__)


adapter = logging.LoggerAdapter(logger,
    {'hello': 'context'}
)

adapter.info('Hello')
```

LogRecordFactory

asyncio context example

```
class ContextLogRecord(logging.LogRecord):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        try:
            request_start = REQUEST_START.get()
        except LookupError:
            REQUEST_PASSED.set(0)
            self.request_passed = 0
        else:
            now = time.monotonic()
            self.request_passed = now - request_start

logging.setLogRecordFactory(ContextLogRecord)
```

Context Logging

https://github.com/afonasev/context_logging

- threads
- asyncio
- ...
- profit

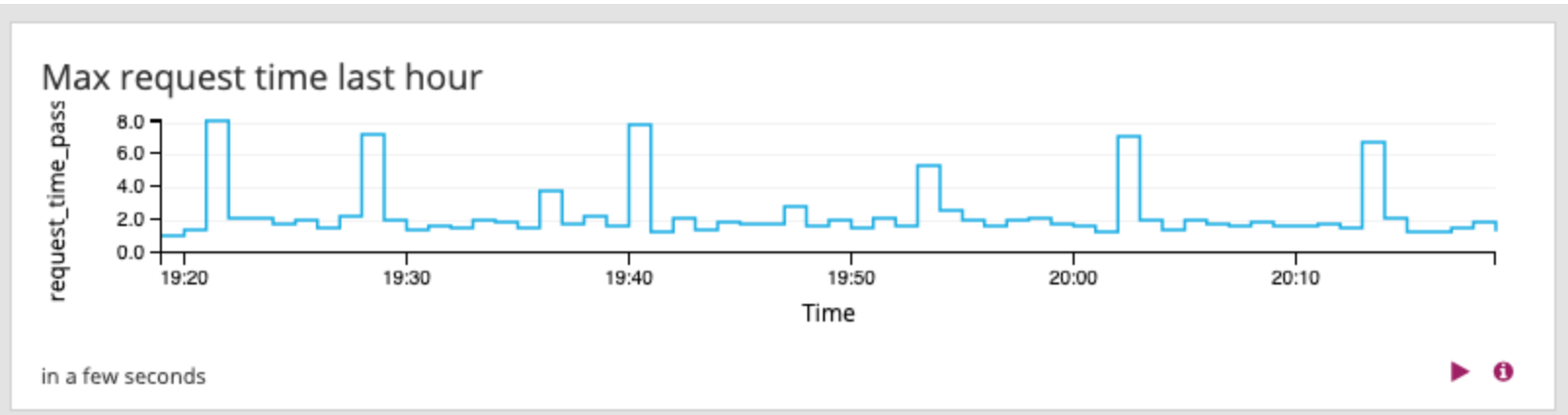
Profiler from Logging

- At the start of the request, store in the context time.monotonic();
- on the next log entry, write the difference between the current moment and the start of the request;
- we get the distribution of what and when happens;
- ...
- profit

Context logging middleware

```
REQUEST_START = contextvars.ContextVar('rqst_start'),

# in middleware on request start

try:
    request_start = REQUEST_START.get()
except LookupError:
    REQUEST_TIME_PASSED.set(0)
else:
    request_now = time.monotonic()
    request_passed = request_now - request_start
    REQUEST_PASSED.set(request_passed)
```

Timestamp	request_time_passed	request_time_start
2019-10-28 20:19:51.881 `Dialog and message created`	0.15599489957094193	23246735.716081843
2019-10-28 20:19:50.911 `Dialog and message created`	0.10434383898973465	23246734.797363024
2019-10-28 20:19:50.752 `Dialog and message created`	0.5425305749522522	965921.722491959
2019-10-28 20:19:50.430 `Dialog and message created`	0.35770679637789726	23250539.661805365
2019-10-28 20:19:50.348 `Dialog and message created`	0.2907202150672674	965921.570245959



Success story

- We can profiling;
- we can investigate;
- we can add alerts.

A little bit about performance

You can flexibly configure in any situation

logger.isEnabledFor(...)

Allows not to call logger at all

```
def some_code(...):
    some_other_code()
    if logger.isEnabledFor(logging.INFO):
        logger.info('all ok')


#---------
log_enable = logger.isEnabledFor(logging.INFO)
def some_code(...):
    some_other_code()
    if log_enable:
        logger.info('all ok')
```

__debug__

Allows not to call logger at all

```
def some_code(...):
    some_other_code()
    if __debug__:
        logger.info('all ok')
```

How configure logging

In modern python, it works without configuration

basicConfig(...)

- It works only once, the second call does **nothing**;
- powerful enough;
- combines with dictConfig.

DictConfig(...)

- Cool for dynamic configuring (from env/config)
- You can initialize handlers and filters

```
"filters": {
  "kafka_in_clear": {
    "()": "FilterRevertSubstringText",
    "substring": "in the clear"
  },
```

```python
class LoggingSettings(BaseSettings):
    level: str = logging.getLevelName(logging.INFO)
    dict_config: Dict[str, Any] = {}

    class Config:
        env_prefix = 'LOG_'

    @validator('dict_config', whole=True)
    def dict_config_validator(cls, v: Dict[str, Any]):
        if not v:
            return None
        return DictConfigurator(v)

    def setup_logging(self) -> None:
        if isinstance(self.dict_config, DictConfigurator):
            self.dict_config.configure()
        else:
            logging.basicConfig(level=self.level)

LoggingSettings().setup_logging()
```

Final idea

- "What I see and write" in the logs
- In stdout only real errors and important information
- We filter out spam and extra messages
- Other message send to graylog | kibana with context

logging-tree

```
>>> import logging_tree
>>> logging_tree.printout()
<--""
   Level WARNING

>>> import concurrent.futures
>>> logging_tree.printout()
<--""
   Level WARNING
   |
   o<--[concurrent]
       |
       o<--"concurrent.futures"
           Level NOTSET so inherits level WARNING
```

loguru

```
from loguru import logger

logger.debug("That's it, beautiful"
             "and simple logging!")
```

Final words

Spasibo

slides.com/kataev/python-logging

kataev

denis.a.kataev@gmail.com