
loguru Documentation

Delgan

Jul 13, 2019

Contents

1 Overview	3
1.1 Installation	3
1.2 Features	3
1.3 Take the tour	4
2 API Reference	11
2.1 loguru.logger	11
3 Resources	27
3.1 Switching from standard logging to loguru	27
3.2 Code snippets and recipes for loguru	31
4 Project Information	39
4.1 Contributing	39
4.2 License	40
4.3 Changelog	41
Python Module Index	45
Index	47

Loguru is a library which aims to bring enjoyable logging in Python.

Did you ever feel lazy about configuring a logger and used `print()` instead?... I did, yet logging is fundamental to every application and eases the process of debugging. Using **Loguru** you have no excuse not to use logging from the start, this is as simple as `from loguru import logger`.

Also, this library is intended to make Python logging less painful by adding a bunch of useful functionalities that solve caveats of the standard loggers. Using logs in your application should be an automatism, **Loguru** tries to make it both pleasant and powerful.

1.1 Installation

```
pip install loguru
```

1.2 Features

- *Ready to use out of the box without boilerplate*
- *No Handler, no Formatter, no Filter: one function to rule them all*
- *Easier file logging with rotation / retention / compression*
- *Modern string formatting using braces style*
- *Exceptions catching within threads or main*
- *Pretty logging with colors*
- *Asynchronous, Thread-safe, Multiprocess-safe*
- *Fully descriptive exceptions*
- *Structured logging as needed*
- *Lazy evaluation of expensive functions*
- *Customizable levels*
- *Better datetime handling*
- *Suitable for scripts and libraries*
- *Entirely compatible with standard logging*
- *Personalizable defaults through environment variables*

- *Convenient parser*
- *Exhaustive notifier*
- *10x faster than built-in logging*

1.3 Take the tour

1.3.1 Ready to use out of the box without boilerplate

The main concept of *Loguru* is that **there is one and only one** `logger`.

For convenience, it is pre-configured and outputs to `stderr` to begin with (but that's entirely configurable).

```
from loguru import logger

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

The `logger` is just an interface which dispatches log messages to configured handlers. Simple, right?

1.3.2 No Handler, no Formatter, no Filter: one function to rule them all

How to add a handler? How to set up logs formatting? How to filter messages? How to set level?

One answer: the `add()` function.

```
logger.add(sys.stderr, format="{time} {level} {message}", filter="my_module", level=
↳ "INFO")
```

This function should be used to register `sinks` which are responsible for managing `log messages` contextualized with a `record dict`. A sink can take many forms: a simple function, a string path, a file-like object, a built-in Handler or a custom class.

Note that you may also `remove()` previously added handlers, this is particularly useful if you want to supersede the default one.

1.3.3 Easier file logging with rotation / retention / compression

If you want to send logged messages to a file, you just have to use a string path as the sink. It can be automatically timed too for convenience:

```
logger.add("file_{time}.log")
```

It is also `easily configurable` if you need rotating logger, if you want to remove older logs, or if you wish to compress your files at closure.

```
logger.add("file_1.log", rotation="500 MB")      # Automatically rotate too big file
logger.add("file_2.log", rotation="12:00")      # New file is created each day at noon
logger.add("file_3.log", rotation="1 week")     # Once the file is too old, it's
↳ rotated

logger.add("file_X.log", retention="10 days")   # Cleanup after some time

logger.add("file_Y.log", compression="zip")    # Save some loved space
```


1.3.4 Modern string formatting using braces style

Loguru favors the much more elegant and powerful `{}` formatting over `%`, logging functions are actually equivalent to `str.format()`.

```
logger.info("If you're using Python {}, prefer {feature} of course!", 3.6, feature="f-
↳strings")
```

1.3.5 Exceptions catching within threads or main

Have you ever seen your program crashing unexpectedly without seeing anything in the log file? Did you ever noticed that exceptions occurring in threads were not logged? This can be solved using the `catch()` decorator / context manager which ensures that any error is correctly propagated to the `logger`.

```
@logger.catch
def my_function(x, y, z):
    # An error? It's caught anyway!
    return 1 / (x + y + z)
```

1.3.6 Pretty logging with colors

Loguru automatically adds colors to your logs if your terminal is compatible. You can define your favorite style by using `markup tags` in the sink format.

```
logger.add(sys.stdout, colorize=True, format="<green>{time}</green> <level>{message}</
↳level>")
```

1.3.7 Asynchronous, Thread-safe, Multiprocess-safe

All sinks added to the `logger` are thread-safe by default. They are not multiprocess-safe, but you can enqueue the messages to ensure logs integrity. This same argument can also be used if you want async logging.

```
logger.add("somefile.log", enqueue=True)
```

1.3.8 Fully descriptive exceptions

Logging exceptions that occur in your code is important to track bugs, but it's quite useless if you don't know why it failed. *Loguru* helps you identify problems by allowing the entire stack trace to be displayed, including values of variables (thanks `better_exceptions` for this!).

The code:

```
logger.add("output.log", backtrace=True, diagnose=True) # Set 'False' to not leak_
↳sensitive data in prod

def func(a, b):
    return a / b

def nested(c):
    try:
        func(5, c)
```

(continues on next page)

(continued from previous page)

```

except ZeroDivisionError:
    logger.exception("What?!")

nested(0)

```

Would result in:

```

2018-07-17 01:38:43.975 | ERROR      | __main__:nested:10 - What?!
Traceback (most recent call last):

  File "test.py", line 12, in <module>
    nested(0)
    L <function nested at 0x7f5c755322f0>

> File "test.py", line 8, in nested
    func(5, c)
    |   L 0
    |   L <function func at 0x7f5c79fc2e18>

  File "test.py", line 4, in func
    return a / b
           L 0
           L 5

ZeroDivisionError: division by zero

```

1.3.9 Structured logging as needed

Want your logs to be serialized for easier parsing or to pass them around? Using the `serialize` argument, each log message will be converted to a JSON string before being sent to the configured sink.

```
logger.add(custom_sink_function, serialize=True)
```

Using `bind()` you can contextualize your logger messages by modifying the `extra` record attribute.

```

logger.add("file.log", format="{extra[ip]} {extra[user]} {message}")
logger_ctx = logger.bind(ip="192.168.0.1", user="someone")
logger_ctx.info("Contextualize your logger easily")
logger_ctx.bind(user="someoneelse").info("Inline binding of extra attribute")

```

You can also have more fine-grained control over your logs by combining `bind()` and `filter`:

```

logger.add("special.log", filter=lambda record: "special" in record["extra"])
logger.debug("This message is not logged to the file")
logger.bind(special=True).info("This message, though, is logged to the file!")

```

Finally, the `patch()` method allows dynamic values to be attached to the record dict of each new message:

```

logger.add(sys.stderr, format="{extra[utc]} {message}")
logger = logger.patch(lambda record: record["extra"].update(utc=datetime.utcnow()))

```

1.3.10 Lazy evaluation of expensive functions

Sometime you would like to log verbose information without performance penalty in production, you can use the `opt()` method to achieve this.

```
logger.opt(lazy=True).debug("If sink level <= DEBUG: {x}", x=lambda: expensive_
↳function(2**64))

# By the way, "opt()" serves many usages
logger.opt(exception=True).info("Error stacktrace added to the log message")
logger.opt(ansi=True).info("Per message <blue>colors</blue>")
logger.opt(record=True).info("Display values from the record (eg. {record[thread]})")
logger.opt(raw=True).info("Bypass sink formatting\n")
logger.opt(depth=1).info("Use parent stack context (useful within wrapped functions)")
```

1.3.11 Customizable levels

Loguru comes with all standard logging levels to which `trace()` and `success()` are added. Do you need more? Then, just create it by using the `level()` function.

```
new_level = logger.level("SNAKY", no=38, color="<yellow>", icon="")
logger.log("SNAKY", "Here we go!")
```

1.3.12 Better datetime handling

The standard logging is bloated with arguments like `datefmt` or `msecs`, `%(asctime)s` and `%(created)s`, naive datetimes without timezone information, not intuitive formatting, etc. *Loguru* fixes it:

```
logger.add("file.log", format="{time:YYYY-MM-DD at HH:mm:ss} | {level} | {message}")
```

1.3.13 Suitable for scripts and libraries

Using the logger in your scripts is easy, and you can `configure()` it at start. To use *Loguru* from inside a library, remember to never call `add()` but use `disable()` instead so logging functions become no-op. If a developer wishes to see your library's logs, he can `enable()` it again.

```
# For scripts
config = {
    "handlers": [
        {"sink": sys.stdout, "format": "{time} - {message}"},
        {"sink": "file.log", "serialize": True},
    ],
    "extra": {"user": "someone"}
}
logger.configure(**config)

# For libraries
logger.disable("my_library")
logger.info("No matter added sinks, this message is not displayed")
logger.enable("my_library")
logger.info("This message however is propagated to the sinks")
```

1.3.14 Entirely compatible with standard logging

Wish to use built-in logging Handler as a *Loguru* sink?

```
handler = logging.handlers.SysLogHandler(address=('localhost', 514))
logger.add(handler)
```

Need to propagate *Loguru* messages to standard *logging*?

```
class PropagateHandler(logging.Handler):
    def emit(self, record):
        logging.getLogger(record.name).handle(record)

logger.add(PropagateHandler(), format="{message}")
```

Want to intercept standard *logging* messages toward your *Loguru* sinks?

```
class InterceptHandler(logging.Handler):
    def emit(self, record):
        # Retrieve context where the logging call occurred, this happens to be in the
        ↪6th frame upward
        logger_opt = logger.opt(depth=6, exception=record.exc_info)
        logger_opt.log(record.levelno, record.getMessage())

logging.basicConfig(handlers=[InterceptHandler()], level=0)
```

1.3.15 Personalizable defaults through environment variables

Don't like the default logger formatting? Would prefer another DEBUG color? No problem:

```
# Linux / OSX
export LOGURU_FORMAT="{time} | <lvl>{message}</lvl>"

# Windows
setx LOGURU_DEBUG_COLOR "<green>"
```

1.3.16 Convenient parser

It is often useful to extract specific information from generated logs, this is why *Loguru* provides a `parse()` method which helps to deal with logs and regexes.

```
pattern = r"(?P<time>.*) - (?P<level>[0-9]+) - (?P<message>.*)" # Regex with named
↪groups
caster_dict = dict(time=dateutil.parser.parse, level=int) # Transform matching
↪groups

for groups in logger.parse("file.log", pattern, cast=caster_dict):
    print("Parsed:", groups)
    # {"level": 30, "message": "Log example", "time": datetime(2018, 12, 09, 11, 23,
    ↪55)}
```

1.3.17 Exhaustive notifier

Loguru can easily be combined with the great `notifiers` library (must be installed separately) to receive an e-mail when your program fail unexpectedly or to send many other kind of notifications.

```
import notifiers

params = {
    "username": "you@gmail.com",
    "password": "abc123",
    "to": "dest@gmail.com"
}

# Send a single notification
notifier = notifiers.get_notifier("gmail")
notifier.notify(message="The application is running!", **params)

# Be alerted on each error message
from notifiers.logging import NotificationHandler

handler = NotificationHandler("gmail", defaults=params)
logger.add(handler, level="ERROR")
```

1.3.18 10x faster than built-in logging

Although logging impact on performances is in most cases negligible, a zero-cost logger would allow to use it anywhere without much concern. In an upcoming release, *Loguru*'s critical functions will be implemented in C for maximum speed.

The Loguru library provides a pre-instanced logger to facilitate dealing with logging in Python.

```
Just from loguru import logger.
```

2.1 loguru.logger

class `Logger`

An object to dispatch logging messages to configured handlers.

The `Logger` is the core object of `loguru`, every logging configuration and usage pass through a call to one of its methods. There is only one logger, so there is no need to retrieve one before usage.

Once the `logger` is imported, it can be used to write messages about events happening in your code. By reading the output logs of your application, you gain a better understanding of the flow of your program and you more easily track and debug unexpected behaviors.

Handlers to which the logger sends log messages are added using the `add()` method. Note that you can use the `Logger` right after import as it comes pre-configured (logs are emitted to `sys.stderr` by default). Messages can be logged with different severity levels and using braces attributes like the `str.format()` method do.

When a message is logged, a “record” is associated with it. This record is a dict which contains information about the logging context: time, function, file, line, thread, level... It also contains the `__name__` of the module, this is why you don’t need named loggers.

You should not instantiate a `Logger` by yourself, use `from loguru import logger` instead.

```
add(sink, *, level='DEBUG', format='<green>{time:YYYY-MM-DD HH:mm:ss.SSS}</green> |  
<level>{level: <8}</level> | <cyan>{name}</cyan>:<cyan>{function}</cyan>:<cyan>{line}</cyan>  
- <level>{message}</level>', filter=None, colorize=None, serialize=False, backtrace=True, diag-  
nose=True, enqueue=False, catch=True, **kwargs)  
Add a handler sending log messages to a sink adequately configured.
```

Parameters

- **sink** (`file-like object`, `str`, `pathlib.Path`, `function`, `logging.Handler` or `class`) – An object in charge of receiving formatted logging messages and propagating them to an appropriate endpoint.
- **level** (`int` or `str`, optional) – The minimum severity level from which logged messages should be sent to the sink.
- **format** (`str` or `function`, optional) – The template used to format logged messages before being sent to the sink.
- **filter** (`function` or `str`, optional) – A directive optionally used to decide for each logged message whether it should be sent to the sink or not.
- **colorize** (`bool`, optional) – Whether the color markups contained in the formatted message should be converted to ansi codes for terminal coloration, or stripped otherwise. If `None`, the choice is automatically made based on the sink being a `tty` or not.
- **serialize** (`bool`, optional) – Whether the logged message and its records should be first converted to a JSON string before being sent to the sink.
- **backtrace** (`bool`, optional) – Whether the exception trace formatted should be extended upward, beyond the catching point, to show the full stacktrace which generated the error.
- **diagnose** (`bool`, optional) – Whether the exception trace should display the variables values to ease the debugging. This should be set to `False` in production to avoid leaking sensitive data.
- **enqueue** (`bool`, optional) – Whether the messages to be logged should first pass through a multiprocessing-safe queue before reaching the sink. This is useful while logging to a file through multiple processes.
- **catch** (`bool`, optional) – Whether errors occurring while sink handles logs messages should be automatically caught. If `True`, an exception message is displayed on `sys.stderr` but the exception is not propagated to the caller, preventing your app to crash.
- ****kwargs** – Additional parameters that will be passed to the sink while creating it or while logging messages (the exact behavior depends on the sink type).

If and only if the sink is a file, the following parameters apply:

Parameters

- **rotation** (`str`, `int`, `datetime.time`, `datetime.timedelta` or `function`, optional) – A condition indicating whenever the current logged file should be closed and a new one started.
- **retention** (`str`, `int`, `datetime.timedelta` or `function`, optional) – A directive filtering old files that should be removed during rotation or end of program.
- **compression** (`str` or `function`, optional) – A compression or archive format to which log files should be converted at closure.
- **delay** (`bool`, optional) – Whether the file should be created as soon as the sink is configured, or delayed until first logged message. It defaults to `False`.
- **mode** (`str`, optional) – The opening mode as for built-in `open()` function. It defaults to `"a"` (open the file in appending mode).
- **buffering** (`int`, optional) – The buffering policy as for built-in `open()` function. It defaults to `1` (line buffered file).

- **encoding** (*str*, optional) – The file encoding as for built-in `open()` function. If `None`, it defaults to `locale.getpreferredencoding()`.
- ****kwargs** – Others parameters are passed to the built-in `open()` function.

Returns `int` – An identifier associated with the added sink and which should be used to `remove()` it.

Notes

Extended summary follows.

The sink parameter

The `sink` handles incoming log messages and proceed to their writing somewhere and somehow. A sink can take many forms:

- A file-like object like `sys.stderr` or `open("somefile.log", "w")`. Anything with a `.write()` method is considered as a file-like object. If it has a `.flush()` method, it will be automatically called after each logged message. If it has a `.stop()` method, it will be automatically called at sink termination.
- A file path as `str` or `pathlib.Path`. It can be parametrized with some additional parameters, see below.
- A simple function like `lambda msg: print(msg)`. This allows for logging procedure entirely defined by user preferences and needs.
- A built-in `logging.Handler` like `logging.StreamHandler`. In such a case, the *Loguru* records are automatically converted to the structure expected by the `logging` module.
- A `class` object that will be used to instantiate the sink using `**kwargs` attributes passed. Hence, the class should instantiate objects which are therefore valid sinks.

Note that you should avoid using the `logger` inside any of your sinks as this would result in infinite recursion or dead lock if the module's sink was not explicitly disabled.

The logged message

The logged message passed to all added sinks is nothing more than a string of the formatted log, to which a special attribute is associated: the `.record` which is a dict containing all contextual information possibly needed (see below).

Logged messages are formatted according to the `format` of the added sink. This format is usually a string containing braces fields to display attributes from the record dict.

If fine-grained control is needed, the `format` can also be a function which takes the record as parameter and return the format template string. However, note that in such a case, you should take care of appending the line ending and exception field to the returned format, while `"\n{exception}"` is automatically appended for convenience if `format` is a string.

The `filter` attribute can be used to control which messages are effectively passed to the sink and which one are ignored. A function can be used, accepting the record as an argument, and returning `True` if the message should be logged, `False` otherwise. If a string is used, only the records with the same name and its children will be allowed.

The severity levels

Each logged message is associated with a severity level. These levels make it possible to prioritize messages and to choose the verbosity of the logs according to usages. For example, it allows to display some debugging information to a developer, while hiding it to the end user running the application.

The `level` attribute of every added sink controls the minimum threshold from which log messages are allowed to be emitted. While using the `logger`, you are in charge of configuring the appropriate granularity of your logs. It is possible to add even more custom levels by using the `level()` method.

Here are the standard levels with their default severity value, each one is associated with a logging method of the same name:

Level name	Severity value	Logger method
TRACE	5	<code>logger.trace()</code>
DEBUG	10	<code>logger.debug()</code>
INFO	20	<code>logger.info()</code>
SUCCESS	25	<code>logger.success()</code>
WARNING	30	<code>logger.warning()</code>
ERROR	40	<code>logger.error()</code>
CRITICAL	50	<code>logger.critical()</code>

The record dict

The record is just a Python dict, accessible from sinks by `message.record`. It contains all contextual information of the logging call (time, function, file, line, level, etc.).

Each of its key can be used in the handler's `format` so the corresponding value is properly displayed in the logged message (e.g. `"{level}"` -> `"INFO"`). Some record's values are objects with two or more attributes, these can be formatted with `"{key.attr}"` (`"{key}"` would display one by default). [Formatting directives](#) like `"{key: >3}"` also works and is particularly useful for time (see below).

Key	Description	Attributes
<code>elapsed</code>	The time elapsed since the start of the program	See <code>datetime.timedelta</code>
<code>exception</code>	The formatted exception if any, None otherwise	<code>type</code> , <code>value</code> , <code>traceback</code>
<code>extra</code>	The dict of attributes bound by the user (see <code>bind()</code>)	None
<code>file</code>	The file where the logging call was made	<code>name</code> (default), <code>path</code>
<code>function</code>	The function from which the logging call was made	None
<code>level</code>	The severity used to log the message	<code>name</code> (default), <code>no</code> , <code>icon</code>
<code>line</code>	The line number in the source code	None
<code>message</code>	The logged message (not yet formatted)	None
<code>module</code>	The module where the logging call was made	None
<code>name</code>	The <code>__name__</code> where the logging call was made	None
<code>process</code>	The process in which the logging call was made	<code>name</code> , <code>id</code> (default)
<code>thread</code>	The thread in which the logging call was made	<code>name</code> , <code>id</code> (default)
<code>time</code>	The aware local time when the logging call was made	See <code>datetime.datetime</code>

The time formatting

To use your favorite time representation, you can set it directly in the time formatter specifier of your handler format, like for example `format="{time:HH:mm:ss} {message}"`. Note that this `datetime`

represents your local time, and it is also made timezone-aware, so you can display the UTC offset to avoid ambiguities.

The time field can be formatted using more human-friendly tokens. These constitute a subset of the one used by the `Pendulum` library of `@sdispater`. To escape a token, just add square brackets around it, for example `"[YY]"` would display literally `"YY"`.

If no time formatter specifier is used, like for example if `format="{time} {message}"`, the default one will use ISO 8601.

	Token	Output
Year	YYYY	2000, 2001, 2002 ... 2012, 2013
	YY	00, 01, 02 ... 12, 13
Quarter	Q	1 2 3 4
Month	MMMM	January, February, March ...
	MMM	Jan, Feb, Mar ...
	MM	01, 02, 03 ... 11, 12
	M	1, 2, 3 ... 11, 12
Day of Year	DDDD	001, 002, 003 ... 364, 365
	DDD	1, 2, 3 ... 364, 365
Day of Month	DD	01, 02, 03 ... 30, 31
	D	1, 2, 3 ... 30, 31
Day of Week	dddd	Monday, Tuesday, Wednesday ...
	ddd	Mon, Tue, Wed ...
	d	0, 1, 2 ... 6
Days of ISO Week	E	1, 2, 3 ... 7
Hour	HH	00, 01, 02 ... 23, 24
	H	0, 1, 2 ... 23, 24
	hh	01, 02, 03 ... 11, 12
	h	1, 2, 3 ... 11, 12
Minute	mm	00, 01, 02 ... 58, 59
	m	0, 1, 2 ... 58, 59
Second	ss	00, 01, 02 ... 58, 59
	s	0, 1, 2 ... 58, 59
Fractional Second	S	0 1 ... 8 9
	SS	00, 01, 02 ... 98, 99
	SSS	000 001 ... 998 999
	SSSS...	000[0..] 001[0..] ... 998[0..] 999[0..]
	SSSSSS	000000 000001 ... 999998 999999
AM / PM	A	AM, PM
Timezone	Z	-07:00, -06:00 ... +06:00, +07:00
	ZZ	-0700, -0600 ... +0600, +0700
	zz	EST CST ... MST PST
Seconds timestamp	X	1381685817, 1234567890.123
Microseconds timestamp	x	1234567890123

The file sinks

If the sink is a `str` or a `pathlib.Path`, the corresponding file will be opened for writing logs. The path can also contain a special `"{time}"` field that will be formatted with the current date at file creation.

The `rotation` check is made before logging each message. If there is already an existing file with the same name that the file to be created, then the existing file is renamed by appending the date to its

basename to prevent file overwriting. This parameter accepts:

- an `int` which corresponds to the maximum file size in bytes before that the current logged file is closed and a new one started over.
- a `datetime.timedelta` which indicates the frequency of each new rotation.
- a `datetime.time` which specifies the hour when the daily rotation should occur.
- a `str` for human-friendly parametrization of one of the previously enumerated types. Examples: "100 MB", "0.5 GB", "1 month 2 weeks", "4 days", "10h", "monthly", "18:00", "sunday", "w0", "monday at 12:00",...
- a `function` which will be called before logging. It should accept two arguments: the logged message and the file object, and it should return `True` if the rotation should happen now, `False` otherwise.

The `retention` occurs at rotation or at sink stop if rotation is `None`. Files are selected according to their basename, if it is the same that the sink file, with possible time field being replaced with `.*`. This parameter accepts:

- an `int` which indicates the number of log files to keep, while older files are removed.
- a `datetime.timedelta` which specifies the maximum age of files to keep.
- a `str` for human-friendly parametrization of the maximum age of files to keep. Examples: "1 week, 3 days", "2 months",...
- a `function` which will be called before the retention process. It should accept the list of log files as argument and process to whatever it wants (moving files, removing them, etc.).

The `compression` happens at rotation or at sink stop if rotation is `None`. This parameter accepts:

- a `str` which corresponds to the compressed or archived file extension. This can be one of: "gz", "bz2", "xz", "lzma", "tar", "tar.gz", "tar.bz2", "tar.xz", "zip".
- a `function` which will be called before file termination. It should accept the path of the log file as argument and process to whatever it wants (custom compression, network sending, removing it, etc.).

The color markups

To add colors to your logs, you just have to enclose your format string with the appropriate tags (e.g. `<tag>some message</tag>`). These tags are automatically removed if the sink doesn't support ansi codes. For convenience, you can use `</>` to close the last opening tag without repeating its name (e.g. `<tag>another message</>`).

The special tag `<level>` (abbreviated with `<lvl>`) is transformed according to the configured color of the logged message level.

Tags which are not recognized will raise an exception during parsing, to inform you about possible misuse. If you wish to display a markup tag literally, you can escape it by prepending a `\` like for example `<blue>`. If, for some reason, you need to escape a string programmatically, note that the regex used internally to parse markup tags is `r"\\?</? ((?: [fb]g\s)? [^<>\s]*)">`.

Here are the available tags (note that compatibility may vary depending on terminal):

Color (abbr)	Styles (abbr)
Black (k)	Bold (b)
Blue (e)	Dim (d)
Cyan (c)	Normal (n)
Green (g)	Italic (i)
Magenta (m)	Underline (u)
Red (r)	Strike (s)
White (w)	Reverse (r)
Yellow (y)	Blink (l)
	Hide (h)

Usage:

Description	Examples	
	Foreground	Background
Basic colors	<red>, <r>	<GREEN>, <G>
Light colors	<light-blue>, <le>	<LIGHT-CYAN>, <LC>
8-bit colors	<fg 86>, <fg 255>	<bg 42>, <bg 9>
Hex colors	<fg #00005f>, <fg #EE1>	<bg #AF5FD7>, <bg #fff>
RGB colors	<fg 0, 95, 0>	<bg 72, 119, 65>
Stylizing	<bold>, , <underline>, <u>	

The environment variables

The default values of sink parameters can be entirely customized. This is particularly useful if you don't like the log format of the pre-configured sink.

Each of the `add()` default parameter can be modified by setting the `LOGURU_[PARAM]` environment variable. For example on Linux: `export LOGURU_FORMAT="{time} - {message}"` or `export LOGURU_DIAGNOSE=NO`.

The default levels' attributes can also be modified by setting the `LOGURU_[LEVEL]_[ATTR]` environment variable. For example, on Windows: `setx LOGURU_DEBUG_COLOR "<blue>"` or `setx LOGURU_TRACE_ICON ""`.

If you want to disable the pre-configured sink, you can set the `LOGURU_AUTOINIT` variable to `False`.

On Linux, you will probably need to edit the `~/.profile` file to make this persistent. On Windows, don't forget to restart your terminal for the change to be taken into account.

Examples

```
>>> logger.add(sys.stdout, format="{time} - {level} - {message}", filter="sub.
↳module")
```

```
>>> logger.add("file_{time}.log", level="TRACE", rotation="100 MB")
```

```
>>> def my_sink(message):
...     record = message.record
...     update_db(message, time=record.time, level=record.level)
...
>>> logger.add(my_sink)
```

```
>>> from logging import StreamHandler
>>> logger.add(StreamHandler(sys.stderr), format="{message}")
```

```
>>> class RandomStream:
...     def __init__(self, seed, threshold):
...         self.threshold = threshold
...         random.seed(seed)
...     def write(self, message):
...         if random.random() > self.threshold:
...             print(message)
...
>>> stream_object = RandomStream(seed=12345, threshold=0.25)
>>> logger.add(stream_object, level="INFO")
>>> logger.add(RandomStream, level="DEBUG", seed=34567, threshold=0.5)
```

remove (*handler_id=None*)

Remove a previously added handler and stop sending logs to its sink.

Parameters *handler_id* (*int* or *None*) – The id of the sink to remove, as it was returned by the *add()* method. If *None*, all handlers are removed. The pre-configured handler is guaranteed to have the index 0.

Raises *ValueError* – If *handler_id* is not *None* but there is no active handler with such id.

Examples

```
>>> i = logger.add(sys.stderr, format="{message}")
>>> logger.info("Logging")
Logging
>>> logger.remove(i)
>>> logger.info("No longer logging")
```

catch (*exception=<class 'Exception'>*, ***, *level='ERROR'*, *reraise=False*, *message="An error has been caught in function '{record[function]}'*, *process '{record[process].name}'* (*{record[process].id}*), *thread '{record[thread].name}'* (*{record[thread].id}*):")

Return a decorator to automatically log possibly caught error in wrapped function.

This is useful to ensure unexpected exceptions are logged, the entire program can be wrapped by this method. This is also very useful to decorate `threading.Thread.run()` methods while using threads to propagate errors to the main logger thread.

Note that the visibility of variables values (which uses the cool `better_exceptions` library from @Qix-) depends on the `diagnose` option of each configured sink.

The returned object can also be used as a context manager.

Parameters

- **exception** (*Exception*, optional) – The type of exception to intercept. If several types should be caught, a tuple of exceptions can be used too.
- **level** (*str* or *int*, optional) – The level name or severity with which the message should be logged.
- **reraise** (*bool*, optional) – Whether the exception should be raised again and hence propagated to the caller.

- **message** (`str`, optional) – The message that will be automatically logged if an exception occurs. Note that it will be formatted with the `record` attribute.

Returns *decorator / context manager* – An object that can be used to decorate a function or as a context manager to log exceptions possibly caught.

Examples

```
>>> @logger.catch
... def f(x):
...     100 / x
...
>>> def g():
...     f(10)
...     f(0)
...
>>> g()
ERROR - An error has been caught in function 'g', process 'Main' (367),
↳thread 'ch1' (1398):
Traceback (most recent call last):
  File "program.py", line 12, in <module>
    g()
    L <function g at 0x7f225fe2bc80>
> File "program.py", line 10, in g
    f(0)
    L <function f at 0x7f225fe2b9d8>
  File "program.py", line 6, in f
    100 / x
    L 0
ZeroDivisionError: division by zero
```

```
>>> with logger.catch(message="Because we never know..."):
...     main() # No exception, no logs
...
...

```

opt (*, *exception=None, record=False, lazy=False, ansi=False, raw=False, depth=0*)

Parametrize a logging call to slightly change generated log message.

Parameters

- **exception** (`bool`, `tuple` or `Exception`, optional) – If it does not evaluate as `False`, the passed exception is formatted and added to the log message. It could be an `Exception` object or a (`type`, `value`, `traceback`) tuple, otherwise the exception information is retrieved from `sys.exc_info()`.
- **record** (`bool`, optional) – If `True`, the record dict contextualizing the logging call can be used to format the message by using `{record[key]}` in the log message.
- **lazy** (`bool`, optional) – If `True`, the logging call attribute to format the message should be functions which will be called only if the level is high enough. This can be used to avoid expensive functions if not necessary.
- **ansi** (`bool`, optional) – If `True`, logged message will be colorized according to the markups it possibly contains.
- **raw** (`bool`, optional) – If `True`, the formatting of each sink will be bypassed and the message will be sent as is.

- **depth** (*int*, optional) – Specify which stacktrace should be used to contextualize the logged message. This is useful while using the logger from inside a wrapped function to retrieve worthwhile information.

Returns *Logger* – A logger wrapping the core logger, but transforming logged message adequately before sending.

Examples

```
>>> try:
...     1 / 0
... except ZeroDivisionError:
...     logger.opt(exception=True).debug("Exception logged with debug level:")
...
[18:10:02] DEBUG in '<module>' - Exception logged with debug level:
Traceback (most recent call last, catch point marked):
> File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero
```

```
>>> logger.opt(record=True).info("Current line is: {record[line]}")
[18:10:33] INFO in '<module>' - Current line is: 1
```

```
>>> logger.opt(lazy=True).debug("If sink <= DEBUG: {x}", x=lambda: math.
↪factorial(2**5))
[18:11:19] DEBUG in '<module>' - If sink <= DEBUG:
↪26313083693369353016721801216000000
```

```
>>> logger.opt(ansi=True).warning("We got a <red>BIG</red> problem")
[18:11:30] WARNING in '<module>' - We got a BIG problem
```

```
>>> logger.opt(raw=True).debug("No formatting\n")
No formatting
```

```
>>> def wrapped():
...     logger.opt(depth=1).info("Get parent context")
...
>>> def func():
...     wrapped()
...
>>> func()
[18:11:54] DEBUG in 'func' - Get parent context
```

bind(***kwargs*)

Bind attributes to the extra dict of each logged message record.

This is used to add custom context to each logging call.

Parameters ****kwargs** – Mapping between keys and values that will be added to the extra dict.

Returns *Logger* – A logger wrapping the core logger, but which sends record with the customized extra dict.

Examples

```
>>> logger.add(sys.stderr, format="{extra[ip]} - {message}")
1
>>> class Server:
...     def __init__(self, ip):
...         self.ip = ip
...         self.logger = logger.bind(ip=ip)
...     def call(self, message):
...         self.logger.info(message)
...
>>> instance_1 = Server("192.168.0.200")
>>> instance_2 = Server("127.0.0.1")
>>> instance_1.call("First instance")
192.168.0.200 - First instance
>>> instance_2.call("Second instance")
127.0.0.1 - Second instance
```

patch (patcher)

Attach a function to modify the record dict created by each logging call.

The `patcher` may be used to update the record on-the-fly before it's propagated to the handlers. This allows the “extra” dict to be populated with dynamic values and also permits advanced modifications of the record emitted while logging a message. The function is called once before sending the log message to the different handlers.

It is recommended to apply modification on the `record["extra"]` dict rather than on the `record` dict itself, as some values are used internally by Loguru, and modify them may produce unexpected results.

Parameters `patcher` (function) – The function to which the record dict will be passed as the sole argument. This function is in charge of updating the record in-place, the function does not need to return any value, the modified record object will be re-used.

Returns `Logger` – A logger wrapping the core logger, but which records are passed through the `patcher` function before being sent to the added handlers.

Examples

```
>>> logger.add(sys.stderr, format="{extra[utc]} {message}")
>>> logger = logger.patch(lambda record: record["extra"].update(utc=datetime.
↳ utcnow()))
>>> logger.info("That's way, you can log messages with time displayed in UTC")
```

```
>>> def wrapper(func):
...     @functools.wraps(func)
...     def wrapped(*args, **kwargs):
...         logger.patch(lambda r: r.update(function=func.__name__)).info(
↳ "Wrapped!")
...         return func(*args, **kwargs)
...     return wrapped
```

```
>>> def recv_record_from_network(pipe):
...     record = pickle.loads(pipe.read())
...     level, message = record["level"], record["message"]
...     logger.patch(lambda r: r.update(record)).log(level, message)
```

level (*name*, *no=None*, *color=None*, *icon=None*)

Add, update or retrieve a logging level.

Logging levels are defined by their name to which a severity `no`, an ansi `color` and an `icon` are associated and possibly modified at run-time. To `log()` to a custom level, you should necessarily use its name, the severity number is not linked back to levels name (this implies that several levels can share the same severity).

To add a new level, all parameters should be passed so it can be properly configured.

To update an existing level, pass its `name` with the parameters to be changed.

To retrieve level information, the `name` solely suffices.

Parameters

- **name** (`str`) – The name of the logging level.
- **no** (`int`) – The severity of the level to be added or updated.
- **color** (`str`) – The color markup of the level to be added or updated.
- **icon** (`str`) – The icon of the level to be added or updated.

Returns `Level` – A namedtuple containing information about the level.

Examples

```
>>> level = logger.level("ERROR")
Level(no=40, color='<red><bold>', icon='')
>>> logger.add(sys.stderr, format="{level.no} {icon} {message}")
>>> logger.level("CUSTOM", no=15, color="<blue>", icon="@")
>>> logger.log("CUSTOM", "Logging...")
15 @ Logging...
>>> logger.level("WARNING", icon=r"/!\")
>>> logger.warning("Updated!")
30 /!\ Updated!
```

disable (*name*)

Disable logging of messages coming from `name` module and its children.

Developers of library using *Loguru* should absolutely disable it to avoid disrupting users with unrelated logs messages.

Note that in some rare circumstances, it is not possible for *Loguru* to determine the module's `__name__` value. In such situation, `record["name"]` will be equals to `None`, this is why `None` is also a valid argument.

Parameters **name** (`str` or `None`) – The name of the parent module to disable.

Examples

```
>>> logger.info("Allowed message by default")
[22:21:55] Allowed message by default
>>> logger.disable("my_library")
>>> logger.info("While publishing a library, don't forget to disable logging")
```

enable (*name*)

Enable logging of messages coming from `name` module and its children.

Logging is generally disabled by imported library using *Loguru*, hence this function allows users to receive these messages anyway.

To enable all logs regardless of the module they are coming from, an empty string "" can be passed.

Parameters `name` (`str` or `None`) – The name of the parent module to re-allow.

Examples

```
>>> logger.disable("__main__")
>>> logger.info("Disabled, so nothing is logged.")
>>> logger.enable("__main__")
>>> logger.info("Re-enabled, messages are logged.")
[22:46:12] Re-enabled, messages are logged.
```

configure (*, `handlers=None`, `levels=None`, `extra=None`, `patch=None`, `activation=None`)

Configure the core logger.

It should be noted that `extra` values set using this function are available across all modules, so this is the best way to set overall default values.

Parameters

- **handlers** (`list` of `dict`, optional) – A list of each handler to be added. The list should contain dicts of params passed to the `add()` function as keyword arguments. If not `None`, all previously added handlers are first removed.
- **levels** (`list` of `dict`, optional) – A list of each level to be added or updated. The list should contain dicts of params passed to the `level()` function as keyword arguments. This will never remove previously created levels.
- **extra** (`dict`, optional) – A dict containing additional parameters bound to the core logger, useful to share common properties if you call `bind()` in several of your files modules. If not `None`, this will remove previously configured `extra` dict.
- **patch** (`function`, optional) – A function that will be applied to the record dict of each logged messages across all modules using the logger. It should modify the dict in-place without returning anything. The function is executed prior to the one possibly added by the `patch()` method. If not `None`, this will replace previously configured `patch` function.
- **activation** (`list` of `tuple`, optional) – A list of (`name`, `state`) tuples which denotes which loggers should be enabled (if `state` is `True`) or disabled (if `state` is `False`). The calls to `enable()` and `disable()` are made accordingly to the list order. This will not modify previously activated loggers, so if you need a fresh start prepend your list with (`"", False`) or (`"", True`).

Returns `list` of `int` – A list containing the identifiers of added sinks (if any).

Examples

```
>>> logger.configure(
...     handlers=[
...         dict(sink=sys.stderr, format="[{time}] {message}"),
...         dict(sink="file.log", enqueue=True, serialize=True),
...     ],
...     levels=[dict(name="NEW", no=13, icon="Q", color="")],
...     extra={"common_to_all": "default"},
```

(continues on next page)

(continued from previous page)

```

...     patch=lambda record: record["extra"].update(some_value=42),
...     activation=[("my_module.secret", False), ("another_library.module",
↳True)],
... )
[1, 2]

```

```

>>> # Set a default "extra" dict to logger across all modules, without "bind()
↳"
>>> extra = {"context": "foo"}
>>> logger.configure(extra=extra)
>>> logger.start(sys.stderr, format="{extra[context]} - {message}")
>>> logger.info("Context without bind")
>>> # => "foo - Context without bind"
>>> logger.bind(context="bar").info("Suppress global context")
>>> # => "bar - Suppress global context"

```

static parse (*file*, *pattern*, *, *cast*={}, *chunk*=65536)

Parse raw logs and extract each entry as a dict.

The logging format has to be specified as the regex *pattern*, it will then be used to parse the *file* and retrieve each entry based on the named groups present in the regex.

Parameters

- **file** (*str*, *pathlib.Path* or *file-like object*) – The path of the log file to be parsed, or an already opened file object.
- **pattern** (*str* or *re.Pattern*) – The regex to use for logs parsing, it should contain named groups which will be included in the returned dict.
- **cast** (*function* or *dict*, optional) – A function that should convert in-place the regex groups parsed (a dict of string values) to more appropriate types. If a dict is passed, it should be a mapping between keys of parsed log dict and the function that should be used to convert the associated value.
- **chunk** (*int*, optional) – The number of bytes read while iterating through the logs, this avoids having to load the whole file in memory.

Yields *dict* – The dict mapping regex named groups to matched values, as returned by *re.Match.groupdict()* and optionally converted according to *cast* argument.

Examples

```

>>> reg = r"(?P<lvl>[0-9]+): (?P<msg>.*)" # If log format is "{level.no} -
↳{message}"
>>> for e in logger.parse("file.log", reg): # A file line could be "10 - A_
↳debug message"
...     print(e) # => {'lvl': '10', 'msg': 'A_
↳debug message'}
...

```

```

>>> caster = dict(lvl=int) # Parse 'lvl' key as an integer
>>> for e in logger.parse("file.log", reg, cast=caster):
...     print(e) # => {'lvl': 10, 'msg': 'A debug_
↳message'}

```

```

>>> def cast(groups):
...     if "date" in groups:
...         groups["date"] = datetime.strptime(groups["date"], "%Y-%m-%d %H:
↪%M:%S")
...
>>> with open("file.log") as file:
...     for log in logger.parse(file, reg, cast=cast):
...         print(log["date"], log["something_else"])

```

trace (*_message*, **args*, ***kwargs*)
 Log *_message*.format(**args*, ***kwargs*) with severity 'TRACE'.

debug (*_message*, **args*, ***kwargs*)
 Log *_message*.format(**args*, ***kwargs*) with severity 'DEBUG'.

info (*_message*, **args*, ***kwargs*)
 Log *_message*.format(**args*, ***kwargs*) with severity 'INFO'.

success (*_message*, **args*, ***kwargs*)
 Log *_message*.format(**args*, ***kwargs*) with severity 'SUCCESS'.

warning (*_message*, **args*, ***kwargs*)
 Log *_message*.format(**args*, ***kwargs*) with severity 'WARNING'.

error (*_message*, **args*, ***kwargs*)
 Log *_message*.format(**args*, ***kwargs*) with severity 'ERROR'.

critical (*_message*, **args*, ***kwargs*)
 Log *_message*.format(**args*, ***kwargs*) with severity 'CRITICAL'.

exception (*_message*, **args*, ***kwargs*)
 Convenience method for logging an 'ERROR' with exception information.

log (*_level*, *_message*, **args*, ***kwargs*)
 Log *_message*.format(**args*, ***kwargs*) with severity *_level*.

start (**args*, ***kwargs*)
 Deprecated function to *add()* a new handler.

Warning: Deprecated since version 0.2.2: *start()* will be removed in Loguru 1.0.0, it is replaced by *add()* which is a less confusing name.

stop (**args*, ***kwargs*)
 Deprecated function to *remove()* an existing handler.

Warning: Deprecated since version 0.2.2: *stop()* will be removed in Loguru 1.0.0, it is replaced by *remove()* which is a less confusing name.

- *Logger*

- *add()*

- * *The sink parameter*
- * *The logged message*
- * *The severity levels*
- * *The record dict*

- * *The time formatting*
- * *The file sinks*
- * *The color markups*
- * *The environment variables*
- *remove()*
- *catch()*
- *opt()*
- *bind()*
- *patch()*
- *level()*
- *disable()*
- *enable()*
- *configure()*
- *parse()*
- *trace()*
- *debug()*
- *info()*
- *success()*
- *warning()*
- *error()*
- *critical()*
- *log()*
- *exception()*

3.1 Switching from standard logging to loguru

3.1.1 Fundamental differences between logging and loguru

Although `loguru` is written “from scratch” and does not rely on standard logging internally, both libraries serve the same purpose: provide functionalities to implement a flexible event logging system. The main difference is that standard logging requires the user to explicitly instantiate named `Logger` and configure them with `Handler`, `Formatter` and `Filter`, while `loguru` tries to narrow down the amount of configuration steps.

Apart from that, usage is globally the same, once the `logger` object is created or imported you can start using it to log messages with the appropriate severity (`logger.debug("Dev message")`, `logger.warning("Danger!")`, etc.), messages which are then sent to the configured handlers.

As for standard logging, default logs are sent to `sys.stderr` rather than `sys.stdout`. The POSIX standard specifies that `stderr` is the correct stream for “diagnostic output”. The main compelling case in favor of logging to `stderr` is that it avoids mixing the actual output of the application with debug information. Consider for example pipe-redirection like `python my_app.py | other_app` which would not be possible if logs were emitted to `stdout`.

3.1.2 Replacing `getLogger()` function

It is usual to call `getLogger()` at the beginning of each file to retrieve and use a logger across your module, like this: `logger = logging.getLogger(__name__)`.

Using `Loguru`, there is no need to explicitly get and name a logger, from `loguru import logger` suffices. Each time this imported logger is used, a *record* is created and will automatically contain the contextual `__name__` value.

As for standard logging, the `name` attribute can then be used to format and filter your logs.

3.1.3 Replacing Logger objects

Loguru replaces the standard `Logger` configuration by a proper *sink* definition. Instead of configuring a logger, you should `add()` and parametrize your handlers. The `setLevel()` and `addFilter()` are suppressed by the configured `level` and `filter` parameters. The `propagate` attribute and `disable()` method can be replaced by the `filter` option too. The `makeRecord()` method can be replaced using the `record["extra"]` dict.

Sometimes, more fine-grained control is required over a particular logger. In such case, Loguru provides the `bind()` method which can be in particular used to generate a specifically named logger.

For example, by calling `other_logger = logger.bind(name="other")`, each *message* logged using `other_logger` will populate the `record["extra"]` dict with the `name` value, while using `logger` won't. This permits differentiating logs from `logger` or `other_logger` from within your sink or filter function.

Let suppose you want a sink to log only some very specific messages:

```
logger.start("specific.log", filter=lambda record: "specific" in record["extra"])

specific_logger = logger.bind(specific=True)

logger.info("General message")           # This is filtered-out by the specific sink
specific_logger.info("Module message")   # This is accepted by the specific sink (and ↪
↪ others)
```

Another example, if you want to attach one sink to one named logger:

```
# Only write messages from "a" logger
logger.start("a.log", filter=lambda record: record["extra"].get("name") == "a")
# Only write messages from "b" logger
logger.start("b.log", filter=lambda record: record["extra"].get("name") == "b")

logger_a = logger.bind(name="a")
logger_b = logger.bind(name="b")

logger_a.info("Message A")
logger_b.info("Message B")
```

3.1.4 Replacing Handler, Filter and Formatter objects

Standard logging requires you to create an `Handler` object and then call `addHandler()`. Using Loguru, the handlers are started using `add()`. The sink defines how the handler should manage incoming logging messages, as would do `handle()` or `emit()`. To log from multiple modules, you just have to import the logger, all messages will be dispatched to the added handlers.

While calling `add()`, the `level` parameter replaces `setLevel()`, the `format` parameter replaces `setFormatter()`, the `filter` parameter replaces `addFilter()`. The thread-safety is managed automatically by Loguru, so there is no need for `createLock()`, `acquire()` nor `release()`. The equivalent method of `removeHandler()` is `remove()` which should be used with the identifier returned by `add()`.

Note that you don't necessarily need to replace your `Handler` objects because `add()` accepts them as valid sinks.

In short, you can replace:

```
logger.setLevel(logging.DEBUG)

fh = logging.FileHandler("spam.log")
fh.setLevel(logging.DEBUG)
```

(continues on next page)

(continued from previous page)

```

ch = logging.StreamHandler()
ch.setLevel(logging.ERROR)

formatter = logging.Formatter("%(asctime)s - %(name)s - %(levelname)s - %(message)s")
fh.setFormatter(formatter)
ch.setFormatter(formatter)

logger.addHandler(fh)
logger.addHandler(ch)

```

With:

```

fmt = "{time} - {name} - {level} - {message}"
logger.add("spam.log", level="DEBUG", format=fmt)
logger.add(sys.stderr, level="ERROR", format=fmt)

```

3.1.5 Replacing % style formatting of messages

Loguru only supports {}-style formatting.

You have to replace `logger.debug("Some variable: %s", var)` with `logger.debug("Some variable: {}", var)`. All `*args` and `**kwargs` passed to a logging function are used to call `message.format(*args, **kwargs)`. Arguments which do not appear in the message string are simply ignored. Note that passing arguments to logging functions like this may be useful to (slightly) improve performances: it avoids formatting the message if the level is too low to pass any configured handler.

For converting the general format used by `Formatter`, refer to *list of available record tokens*.

For converting the date format used by `datefmt`, refer to *list of available date tokens*.

3.1.6 Replacing `exc_info` argument

While calling standard logging function, you can pass `exc_info` as an argument to add stacktrace to the message. Instead of that, you should use the `opt()` method with `exception` parameter, replacing `logger.debug("Debug error:", exc_info=True)` with `logger.opt(exception=True).debug("Debug error:")`.

The formatted exception will include the whole stacktrace and variables. To prevent that, make sure to use `backtrace=False` and `diagnose=False` while adding your sink.

3.1.7 Replacing `extra` argument and `LoggerAdapter` objects

To pass contextual information to log messages, replace `extra` by inlining `bind()` method:

```

context = {"clientip": "192.168.0.1", "user": "fbloggs"}

logger.info("Protocol problem", extra=context)    # Standard logging
logger.bind(**context).info("Protocol problem")  # Loguru

```

This will add context information to the `record["extra"]` dict of your logged message, so make sure to configure your handler format adequately:

```

fmt = "%(asctime)s %(clientip)s %(user)s %(message)s"    # Standard logging
fmt = "{time} {extra[clientip]} {extra[user]} {message}" # Loguru

```

You can also replace `LoggerAdapter` by calling `logger = logger.bind(clientip="192.168.0.1")` before using it, or by assigning the bound logger to a class instance:

```
class MyClass:

    def __init__(self, clientip):
        self.logger = logger.bind(clientip=clientip)

    def func(self):
        self.logger.debug("Running func")
```

3.1.8 Replacing `isEnabledFor()` method

If you wish to log useful information for your debug logs, but don't want to pay the performance penalty in release mode while no debug handler is configured, standard logging provides the `isEnabledFor()` method:

```
if logger.isEnabledFor(logging.DEBUG):
    logger.debug("Message data: %s", expensive_func())
```

You can replace this with the `opt()` method and lazy option:

```
# Arguments should be functions which will be called if needed
logger.opt(lazy=True).debug("Message data: {}", expensive_func)
```

3.1.9 Replacing `addLevelName()` and `getLevelName()` functions

To add a new custom level, you can replace `addLevelName()` with the `level()` function:

```
logging.addLevelName(33, "CUSTOM") # Standard logging
logger.level("CUSTOM", no=45, color="<red>", icon="") # Loguru
```

The same function can be used to replace `getLevelName()`:

```
logger.getLevelName(33) # => "CUSTOM"
logger.level("CUSTOM") # => (no=33, color="<red>", icon="")
```

Note that contrary to standard logging, Loguru doesn't associate severity number to any level, levels are only identified by their name.

3.1.10 Replacing `basicConfig()` and `dictConfig()` functions

The `basicConfig()` and `dictConfig()` functions are replaced by the `configure()` method.

This does not accept `config.ini` files, though, so you have to handle that yourself using your favorite format.

3.1.11 Making things work with Pytest and caplog

`pytest` is a very common testing framework. The `caplog` fixture captures logging output so that it can be tested against. For example:

```
# `some_func` adds two numbers, and logs a warning if the first is < 1
def test_some_func_logs_warning(caplog):
    assert some_func(-1, 3) == 2
    assert "Oh no!" in caplog.text
```

If you've followed all the migration guidelines thus far, you'll notice that this test will fail. This is because `pytest` links to the standard library's logging module.

So to fix things, we need to add a sink that propagates Loguru to logging. This is done on the fixture itself by monkeypatching `caplog`. In your `conf/test.py` file, add the following:

```
import logging
import pytest
from _pytest.logging import caplog as _caplog
from loguru import logger

@pytest.fixture
def caplog(_caplog):
    class PropagateHandler(logging.Handler):
        def emit(self, record):
            logging.getLogger(record.name).handle(record)

    handler_id = logger.add(PropagateHandler(), format="{message}")
    yield _caplog
    logger.remove(handler_id)
```

Run your tests and things should all be working as expected. Additional information can be found in [GH#59](#).

3.2 Code snippets and recipes for loguru

3.2.1 Changing the level of an existing handler

Once a handler has been added, it is actually not possible to update it. This is a deliberate choice in order to keep the Loguru's API minimal. Several solutions are possible, though, if you need to change the configured level of a handler. Chose the one that best fits your use case.

The most straightforward workaround is to `remove()` your handler and then `re-add()` it with the updated level parameter. To do so, you have to keep a reference to the identifier number returned while adding a handler:

```
handler_id = logger.add(sys.stderr, level="WARNING")

logger.info("Logging 'WARNING' or higher messages only")

...

logger.remove(handler_id)
logger.add(sys.stderr, level="DEBUG")

logger.debug("Logging 'DEBUG' messages too")
```

Alternatively, you can combine the `bind()` method with the `filter` argument to provide a function dynamically filtering logs based on their level:

```
def my_filter(record):
    if record["extra"].get("warn_only"): # "warn_only" is bound to the logger and
    ↪ set to 'True'
```

(continues on next page)

(continued from previous page)

```
        return record["level"].no >= logger.level("WARNING").no
    return True # Fallback to default 'level' configured while adding the handler

logger.add(sys.stderr, filter=my_filter, level="DEBUG")

# Use this logger first, debug messages are filtered out
logger = logger.bind(warn_only=True)
logger.warn("Initialization in progress")

# Then you can use this one to log all messages
logger = logger.bind(warn_only=False)
logger.debug("Back to debug messages")
```

Finally, more advanced control over handler's level can be achieved by using a callable object as the filter:

```
class MyFilter:

    def __init__(self, level):
        self.level = level

    def __call__(self, record):
        levelno = logger.level(self.level).no
        return record["level"].no >= levelno

my_filter = MyFilter("WARNING")
logger.add(sys.stderr, filter=my_filter, level=0)

logger.warning("OK")
logger.debug("NOK")

my_filter.level = "DEBUG"
logger.debug("OK")
```

3.2.2 Sending and receiving log messages across network or processes

It is possible to transmit logs between different processes and even between different computer if needed. Once the connection is established between the two Python programs, this requires serializing the logging record in one side while re-constructing the message on the other hand.

This can be achieved using a custom sink for the client and `patch()` for the server.

```
# client.py
import sys
import socket
import struct
import time
import pickle

from loguru import logger

class SocketHandler:
```

(continues on next page)

(continued from previous page)

```

self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
self.sock.connect((host, port))

def write(self, message):
    record = message.record
    data = pickle.dumps(record)
    slen = struct.pack(">L", len(data))
    self.sock.send(slen + data)

logger.configure(handlers=[{"sink": SocketHandler('localhost', 9999)}])

while 1:
    time.sleep(1)
    logger.info("Sending message from the client")

```

```

# server.py
import socketserver
import pickle
import struct

from loguru import logger

class LoggingStreamHandler(socketserver.StreamRequestHandler):

    def handle(self):
        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))
            record = pickle.loads(chunk)
            level, message = record["level"], record["message"]
            logger.patch(lambda record: record.update(record)).log(level, message)

server = socketserver.TCPServer(('localhost', 9999), LoggingStreamHandler)
server.serve_forever()

```

Keep in mind though that [pickling](#) is unsafe, use this with care.

3.2.3 Logging entry and exit of functions with a decorator

In some cases, it might be useful to log entry and exit values of a function. Although Loguru doesn't provide such feature out of the box, it can be easily implemented by using Python decorators:

```

import functools
from loguru import logger

def logger_wraps(*, entry=True, exit=True, level="DEBUG"):

    def wrapper(func):

```

(continues on next page)

(continued from previous page)

```

name = func.__name__

@functools.wraps(func)
def wrapped(*args, **kwargs):
    logger_ = logger.opt(depth=1)
    if entry:
        logger_.log(level, "Entering '{}' (args={}, kwargs={})", name, args,
↳kwargs)
    result = func(*args, **kwargs)
    if exit:
        logger_.log(level, "Exiting '{}' (result={})", name, result)
    return result

    return wrapped

return wrapper

```

You could then use it like this:

```

@logger_wlaps()
def foo(a, b, c):
    logger.info("Inside the function")
    return a * b * c

def bar():
    foo(2, 4, c=8)

bar()

```

Which would result in:

```

2019-04-07 11:08:44.198 | DEBUG      | __main__:bar:30 - Entering 'foo' (args=(2, 4),
↳kwargs={'c': 8})
2019-04-07 11:08:44.198 | INFO       | __main__:foo:26 - Inside the function
2019-04-07 11:08:44.198 | DEBUG      | __main__:bar:30 - Exiting 'foo' (result=64)

```

Here is another simple example to record timing of a function:

```

def timeit(func):

    def wrapped(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        logger.debug("Function '{}' executed in {:.f} s", func.__name__, end - start)
        return result

    return wrapped

```

3.2.4 Using logging function based on custom added levels

After adding a new level, it's habitually used with the `log()` function:

```
logger.level("foobar", no=33, icon="", color="<blue>")

logger.log("foobar", "A message")
```

For convenience, one can assign a new logging function which automatically uses the custom added level:

```
def foobar(_, message, *args, **kwargs):
    logger.opt(depth=1).log("foobar", message, *args, **kwargs)

logger.__class__.foobar = foobar

logger.foobar("A message")
```

The new method need to be added only once and will be usable across all your files importing the `logger`. Note that the call to `opt(depth=1)` is necessary to make sure that the logged message contains contextual information of the parent stack frame (where `logger.foobar()` is called). Also, assigning the method to `logger.__class__` rather than `logger` directly ensures that it stays available even after calling `logger.bind()`, `logger.patch()` and `logger.opt()` (because these functions return a new `logger` instance).

3.2.5 Dynamically formatting messages to properly align values with padding

The default formatter is unable to vertically align log messages because the length of `{name}`, `{function}` and `{line}` are not fixed.

One workaround consists of using padding with some maximum value that should suffice most of the time, like this for example:

```
fmt = "{time} | {level: <8} | {name: ^15} | {function: ^15} | {line: >3} | {message}"
logger.add(sys.stderr, format=fmt)
```

Others solutions are possible by using a formatting function or class. For example, it is possible to dynamically adjust the padding length based on previously encountered values:

```
class Formatter:

    def __init__(self):
        self.padding = 0
        self.fmt = "{time} | {level: <8} | {name}:{function}:{line}{extra[padding]} |
↪{message}\n(exception)"

    def format(self, record):
        length = len("{name}:{function}:{line}".format(**record))
        self.padding = max(self.padding, length)
        record["extra"]["padding"] = " " * (self.padding - length)
        return self.fmt

formatter = Formatter()

logger.remove()
logger.add(sys.stderr, format=formatter.format)
```

3.2.6 Capturing standard stdout, stderr and warnings

The use of logging should be privileged over `print()`, yet, it may happen that you don't have plain control over code executed in your application. If you wish to capture standard output, you can suppress `sys.stdout` (and `sys.`

`stderr`) with a custom stream object. You have to take care of first removing the default handler, and not adding a new stdout sink once redirected or that would cause dead lock (you may use `sys.__stdout__` instead):

```
import sys
from loguru import logger

class StreamToLogger:

    def __init__(self, level="INFO"):
        self._level = level

    def write(self, buffer):
        for line in buffer.rstrip().splitlines():
            logger.log(self._level, line.rstrip())

    def flush(self):
        pass

logger.remove()
logger.add(sys.__stdout__)

sys.stdout = StreamToLogger()
```

You may also capture warnings emitted by your application by replacing `warnings.showwarning()`:

```
import warnings
from loguru import logger

showwarning_ = warnings.showwarning

def showwarning(message, *args, **kwargs):
    logger.warning(message)
    showwarning_(message, *args, **kwargs)

warnings.showwarning = showwarning
```

3.2.7 Circumventing modules whose `__name__` value is absent

Loguru makes use of the global variable `__name__` to determine from where the logged message is coming from. However, it may happen in very specific situation (like some Dask distributed environment) that this value is not set. In such case, Loguru will use `None` to make up for the lack of the value. This implies that if you want to `disable()` messages coming from such special module, you have to explicitly call `logger.disable(None)`.

Similar considerations should be taken into account while dealing with the `filter` attribute. As `__name__` is missing, Loguru will assign the `None` value to the `record["name"]` entry. It also means that once formatted in your log messages, the `{name}` token will be equals to `"None"`. This can be worked around by manually overriding the `record["name"]` value using `patch()` from inside the faulty module:

```
# If Loguru fails to retrieve the proper "name" value, assign it manually
logger = logger.patch(lambda record: record.update(name="my_module"))
```

You probably should not worry about all of this except if you noticed that your code is subject to this behavior.

3.2.8 Using Loguru's `logger` within a Cython module

Loguru and Cython do not interoperate very well. This is because Loguru (and logging generally) heavily relies on Python stack frames while Cython, being an alternative Python implementation, try to get rid of these frames for optimization reasons.

Calling the `logger` from code compiled with Cython may raise this kind of exception:

```
ValueError: call stack is not deep enough
```

This error happens when Loguru tries to access a stack frame which has been suppressed by Cython. There is no way for Loguru to retrieve contextual information of the logged message, but there exists a workaround that will at least prevent your application to crash:

```
# Add this at the start of your file  
logger = logger.opt(depth=-1)
```

Note that logged messages should be displayed correctly, but function name and other information will be incorrect. This issue is discussed in [GH#88](#).

4.1 Contributing

Thank you for considering improving *Loguru*, any contribution is much welcome!

4.1.1 Asking questions

If you have any question about *Loguru*, if you are seeking for help, or if you would like to suggest a new feature, you are encouraged to [open a new issue](#) so we can discuss it. Bringing new ideas and pointing out elements needing clarification allows to make this library always better!

4.1.2 Reporting a bug

If you encountered an unexpected behavior using *Loguru*, please [open a new issue](#) and describe the problem you have spotted. Be as specific as possible in the description of the trouble so we can easily analyse it and quickly fix it.

An ideal bug report includes:

- The Python version you are using
- The *Loguru* version you are using (you can find it with `print(loguru.__version__)`)
- Your operating system name and version (Linux, MacOS, Windows)
- Your development environment and local setup (IDE, Terminal, project context, any relevant information that could be useful)
- Some [minimal reproducible example](#)

4.1.3 Implementing changes

If you are willing to enhance *Loguru* by implementing non-trivial changes, please [open a new issue](#) first to keep a reference about why such modifications are made (and potentially avoid unneeded work). Then, the workflow would look as follows:

1. Fork the [Loguru](#) project from GitHub
2. Clone the repository locally:

```
$ git clone git@github.com:your_name_here/loguru.git
$ cd loguru
```

3. Activate your virtual environment:

```
$ python -m virtualenv env
$ source env/bin/activate
```

4. Create a new branch from master:

```
$ git checkout master
$ git branch fix_bug
$ git checkout fix_bug
```

5. Install *Loguru* in development mode:

```
$ pip install -e .[dev]
```

6. Implement the modifications wished. During the process of development, honor [PEP 8](#) as much as possible.
7. Add unit tests (don't hesitate to be exhaustive!) and ensure none are failing using:

```
$ tox
```

8. Remember to update documentation if required
9. Update the `CHANGELOG.rst` file with what you improved
10. add and commit your changes, rebase your branch on master, push your local project:

```
$ git add .
$ git commit -m 'Add succinct explanation of what changed'
$ git rebase master
$ git push origin fix_bug
```

11. Finally, [open a pull request](#) before getting it merged!

4.2 License

MIT License

Copyright (c) 2017

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

4.3 Changelog

4.3.1 0.3.1 (2019-07-13)

- Fix retention and rotation issues when file sink initialized with `delay=True` (#113).
- Fix "sec" no longer recognized as a valid duration unit for file rotation and retention arguments.
- Ensure stack from the caller is displayed while formatting exception of a function decorated with `@logger.catch when backtrace=False`.
- Modify datetime used to automatically rename conflicting file when rotating (it happens if file already exists because "{time}" not presents in filename) so it's based on the file creation time rather than the current time.

4.3.2 0.3.0 (2019-06-29)

- Remove all dependencies previously needed by loguru (on Windows platform, it solely remains `colorama` and `win32-setctime`).
- Add a new `logger.patch()` method which can be used to modify the record dict on-the-fly before it's being sent to the handlers.
- Modify behavior of sink option `backtrace` so it only extends the `stacktrace` upward, the display of variables values is now controlled with the new `diagnose` argument (#49).
- Change behavior of `rotation` option in file sinks: it is now based on the file creation time rather than the current time, note that proper support may differ depending on your platform (#58).
- Raise errors on unknowns color tags rather than silently ignoring them (#57).
- Add the possibility to auto-close color tags by using `</>` (e.g. `<yellow>message</>`).
- Add coloration of exception traceback even if `diagnose` and `backtrace` options are `False`.
- Add a way to limit the depth of formatted exceptions traceback by setting the conventional `sys.tracebacklimit` variable (#77).
- Add `__repr__` value to the logger for convenient debugging (#84).
- Remove colors tags mixing directives (e.g. `<red,blue>`) for simplification.
- Make the `record["exception"]` attribute unpackable as a `(type, value, traceback)` tuple.
- Fix error happening in some rare circumstances because `frame.f_globals` dict did not contain `"__name__"` key and hence prevented Loguru to retrieve the module's name. From now, `record["name"]` will be equal to `None` in such case (#62).
- Fix logging methods not being serializable with `pickle` and hence raising exception while being passed to some multiprocessing functions (#102).

- Fix exception stack trace not colorizing source code lines on Windows.
- Fix possible `AttributeError` while formatting exceptions within a `celery` task (#52).
- Fix `logger.catch` decorator not working with generator and coroutine functions (#75).
- Fix `record["path"]` case being normalized for no necessary reason (#85).
- Fix some Windows terminal emulators (mintty) not correctly detected as supporting colors, causing ansi codes to be automatically stripped (#104).
- Fix handler added with `enqueue=True` stopping working if exception was raised in sink although `catch=True`.
- Fix thread-safety of `enable()` and `disable()` being called during logging.
- Use `Tox` to run tests (#41).

4.3.3 0.2.5 (2019-01-20)

- Modify behavior of sink option `backtrace=False` so it doesn't extend traceback upward automatically (#30).
- Fix import error on some platforms using Python 3.5 with limited `localtime()` support (#33).
- Fix incorrect time formatting of locale month using `MMM` and `MMMM` tokens (#34, thanks @nasyxx).
- Fix race condition permitting writing on a stopped handler.

4.3.4 0.2.4 (2018-12-26)

- Fix adding handler while logging which was not thread-safe (#22).

4.3.5 0.2.3 (2018-12-16)

- Add support for PyPy.
- Add support for Python 3.5.
- Fix incompatibility with `awscli` by downgrading required `colorama` dependency version (#12).

4.3.6 0.2.2 (2018-12-12)

- Deprecate `logger.start()` and `logger.stop()` methods in favor of `logger.add()` and `logger.remove()` (#3).
- Fix ignored formatting while using `logging.Handler` sinks (#4).
- Fix impossibility to set empty environment variable `color` on Windows (#7).

4.3.7 0.2.1 (2018-12-08)

- Fix typo preventing README to be correctly displayed on PyPI.

4.3.8 0.2.0 (2018-12-08)

- Remove the `parser` and refactor it into the `logger.parse()` method.
- Remove the `notifier` and its dependencies (`pip install notifiers` should be used instead).

4.3.9 0.1.0 (2018-12-07)

- Add `logger`.
- Add `notifier`.
- Add `parser`.

4.3.10 0.0.1 (2017-09-04)

Initial release.

|
loguru, [11](#)

A

`add()` (*Logger method*), 11

B

`bind()` (*Logger method*), 20

C

`catch()` (*Logger method*), 18

`configure()` (*Logger method*), 23

`critical()` (*Logger method*), 25

D

`debug()` (*Logger method*), 25

`disable()` (*Logger method*), 22

E

`enable()` (*Logger method*), 22

`error()` (*Logger method*), 25

`exception()` (*Logger method*), 25

I

`info()` (*Logger method*), 25

L

`level()` (*Logger method*), 21

`log()` (*Logger method*), 25

`Logger` (*class in loguru._logger*), 11

`loguru` (*module*), 11

O

`opt()` (*Logger method*), 19

P

`parse()` (*Logger static method*), 24

`patch()` (*Logger method*), 21

R

`remove()` (*Logger method*), 18

S

`start()` (*Logger method*), 25

`stop()` (*Logger method*), 25

`success()` (*Logger method*), 25

T

`trace()` (*Logger method*), 25

W

`warning()` (*Logger method*), 25