loguru Documentation

Delgan

CONTENTS

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.

ONE

OVERVIEW

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).

```python
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.

```python
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 coroutine function or a built-in Handler.

Note that you may also `remove()` a previously added handler by using the identifier returned while adding it. This is particularly useful if you want to supersede the default `stderr` handler: just call `logger.remove()` to make a fresh start.

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:

```python
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.

```python
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 notice 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)
```

Coroutine functions used as sinks are also supported and should be awaited with `complete()`.

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:

```
# Caution, "diagnose=True" is the default and may leak sensitive data in prod
logger.add("out.log", backtrace=True, diagnose=True)

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

(continues on next page)

```python
def nested(c):
    try:
        func(5, c)
    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)
    └ <function nested at 0x7f5c755322f0>

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

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

ZeroDivisionError: division by zero
```

Note that this feature won't work on default Python REPL due to unavailable frame data.

See also: Security considerations when using Loguru.

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.

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

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

```python
logger.add("file.log", format="{extra[ip]} {extra[user]} {message}")
context_logger = logger.bind(ip="192.168.0.1", user="someone")
context_logger.info("Contextualize your logger easily")
context_logger.bind(user="someone_else").info("Inline binding of extra attribute")
context_logger.info("Use kwargs to add context during formatting: {user}", user="anybody
→")
```

It is possible to modify a context-local state temporarily with `contextualize()`:

```
with logger.contextualize(task=task_id):
    do_something()
    logger.info("End of task")
```

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 (tuple
→accepted too)")
logger.opt(colors=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)")
logger.opt(capture=False).info("Keyword arguments not added to {dest} dict", dest="extra
→")
```

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, they can `enable()` it again.

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

# For libraries, should be your library's `__name__`
logger.disable("my_library")
logger.info("No matter added sinks, this message is not displayed")

# In your application, enable the logger in the library
logger.enable("my_library")
logger.info("This message however is propagated to the sinks")
```

For additional convenience, you can also use the `loguru-config` library to setup the `logger` directly from a configuration file.

1.3.14 Entirely compatible with standard logging

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

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

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

```python
class PropagateHandler(logging.Handler):
    def emit(self, record: logging.LogRecord) -> None:
        logging.getLogger(record.name).handle(record)

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

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

```python
class InterceptHandler(logging.Handler):
    def emit(self, record: logging.LogRecord) -> None:
        # Get corresponding Loguru level if it exists.
        level: str | int
        try:
            level = logger.level(record.levelname).name
        except ValueError:
            level = record.levelno

        # Find caller from where originated the logged message.
        frame, depth = inspect.currentframe(), 0
        while frame and (depth == 0 or frame.f_code.co_filename == logging.__file__):
            frame = frame.f_back
            depth += 1

        logger.opt(depth=depth, exception=record.exc_info).log(level, record.
→getMessage())

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

1.3.15 Personalizable defaults through environment variables

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

```bash
# 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.

```python
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.

```python
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.

API REFERENCE

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 they can be formatted using curly braces (it uses `str.format()` under the hood).

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*, *diagnose=True*, *enqueue=False*, *context=None*, *catch=True*, ***kwargs*)

Add a handler sending log messages to a sink adequately configured.

> **Parameters**
>
> - **sink** (`file-like object`, `str`, `pathlib.Path`, `callable`, `coroutine function` or `logging.Handler`) – 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 `callable`, optional) – The template used to format logged messages before being sent to the sink.

- **filter** (`callable`, `str` or `dict`, 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 eases 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. This also has the advantage of making logging calls non-blocking.

- **context** (`multiprocessing.Context` or `str`, optional) – A context object or name that will be used for all tasks involving internally the `multiprocessing` module, in particular when enqueue=True. If `None`, the default context is used.

- **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 are only valid to configure a coroutine or file sink (see below).

If and only if the sink is a coroutine function, the following parameter applies:

Parameters
> **loop** (`AbstractEventLoop`, optional) – The event loop in which the asynchronous logging task will be scheduled and executed. If `None`, the loop used is the one returned by `asyncio.get_running_loop()` at the time of the logging call (task is discarded if there is no loop currently running).

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

Parameters

- **rotation** (`str`, `int`, `datetime.time`, `datetime.timedelta` or `callable`, optional) – A condition indicating whenever the current logged file should be closed and a new one started.

- **retention** (`str`, `int`, `datetime.timedelta` or `callable`, optional) – A directive filtering old files that should be removed during rotation or end of program.

- **compression** (`str` or `callable`, 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`.

- **watch** (`bool`, optional) – Whether or not the file should be watched and re-opened when deleted or changed (based on its device and inode properties) by an external program. 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. It defaults to `"utf8"`.

- ****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.

Raises

`ValueError` – If any of the arguments passed to configure the sink is invalid.

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("file.log", "w")`. Anything with a `.write()` method is considered as a file-like object. Custom handlers may also implement `flush()` (called after each logged message), `stop()` (called at sink termination) and `complete()` (awaited by the eponymous method).

- A file path as `str` or `pathlib.Path`. It can be parametrized with some additional parameters, see below.

- A `callable` (such as a simple function) like `lambda msg: print(msg)`. This allows for logging procedure entirely defined by user preferences and needs.

- A asynchronous `coroutine function` defined with the `async def` statement. The coroutine object returned by such function will be added to the event loop using `loop.create_task()`. The tasks should be awaited before ending the loop by using *complete()*.

- 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.

Note that the logging functions are not reentrant. This means you should avoid using the `logger` inside any of your sinks or from within `signal` handlers. Otherwise, you may face deadlock 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. One can also pass a `dict` mapping module names to minimum required level. In such case, each log record will search for it's closest parent in the `dict` and use the associated level as the filter. The `dict` values can be `int` severity, `str` level name or `True` and `False` to respectively authorize and discard all module logs unconditionally. In order to set a default level, the `""` module name should be used as it is the parent of all modules (it does not suppress global `level` threshold, though).

Note that while calling a logging method, the keyword arguments (if any) are automatically added to the `extra` dict for convenient contextualization (in addition to being used for formatting).

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	*logger.trace()*
DEBUG	10	*logger.debug()*
INFO	20	*logger.info()*
SUCCESS	25	*logger.success()*
WARNING	30	*logger.warning()*
ERROR	40	*logger.error()*
CRITICAL	50	*logger.critical()*

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 the record keys can be used in the handler's `format` so the corresponding value is properly displayed in the logged message (e.g. `"{level}"` will return `"INFO"`). Some records' values are objects with two or more attributes. These can be formatted with `"{key.attr}"` (`"{key}"` would display one by default).

Note that you can use any formatting directives available in Python's `str.format()` method (e.g. `"{key:>3}"` will right-align and pad to a width of 3 characters). This is particularly useful for time formatting (see below).

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

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 you prefer to display UTC rather than local time, you can add `"!UTC"` at the very end of the time format, like `{time:HH:mm:ss!UTC}`. Doing so will convert the `datetime` to UTC before formatting.

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

continues on next page

Table 1 – continued from previous page

	Token	Output
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 file is closed at sink stop, i.e. when the application ends or the handler is removed.

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 `callable` which will be invoked 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 resulting from previous sessions or rotations are automatically collected from disk. A file is selected if it matches the pattern `"basename(.*).ext(.*)"` (possible time fields are beforehand replaced with `.*`) based on the configured sink. Afterwards, the list is processed to determine files to be retained. This parameter accepts:

- an `int` which indicates the number of log files to keep, while older files are deleted.

- 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 `callable` which will be invoked 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 `callable` which will be invoked 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.).

Either way, if you use a custom function designed according to your preferences, you must be very careful not to use the `logger` within your function. Otherwise, there is a risk that your program hang because of a deadlock.

The color markups

To add colors to your logs, you just have to enclose your format string with the appropriate tags (e.g. `<red>some message</red>`). 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. `<red>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]*)>"`.

Note that when logging a message with `opt(colors=True)`, color tags present in the formatting arguments (`args` and `kwargs`) are completely ignored. This is important if you need to log strings containing markups that might interfere with the color tags (in this case, do not use f-string).

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 (v)
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>`, `<b>`, `<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 use the `set` command, do not include quotes but escape special symbol as needed, e.g. `set LOGURU_DEBUG_COLOR=^<blue^>`.

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 debug_only(record):
...     return record["level"].name == "DEBUG"
...
>>> logger.add("debug.log", filter=debug_only)  # Other levels are filtered out
```

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

```
>>> level_per_module = {
...     "": "DEBUG",
```

(continues on next page)

```
...         "third.lib": "WARNING",
...         "anotherlib": False
... }
>>> logger.add(lambda m: print(m, end=""), filter=level_per_module, level=0)
```

```
>>> async def publish(message):
...         await api.post(message)
...
>>> logger.add(publish, serialize=True)
```

```
>>> 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")
```

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")
>    ```

complete()

>   Wait for the end of enqueued messages and asynchronous tasks scheduled by handlers.

>   This method proceeds in two steps: first it waits for all logging messages added to handlers with
>   enqueue=True to be processed, then it returns an object that can be awaited to finalize all logging tasks
>   added to the event loop by coroutine sinks.

>   It can be called from non-asynchronous code. This is especially recommended when the logger is utilized
>   with multiprocessing to ensure messages put to the internal queue have been properly transmitted before
>   leaving a child process.

The returned object should be awaited before the end of a coroutine executed by `asyncio.run()` or `loop.run_until_complete()` to ensure all asynchronous logging messages are processed. The function `asyncio.get_running_loop()` is called beforehand, only tasks scheduled in the same loop that the current one will be awaited by the method.

> **Returns**
>> awaitable – An awaitable object which ensures all asynchronous logging calls are completed when awaited.

Examples

```
>>> async def sink(message):
...     await asyncio.sleep(0.1)  # IO processing...
...     print(message, end="")
...
>>> async def work():
...     logger.info("Start")
...     logger.info("End")
...     await logger.complete()
...
>>> logger.add(sink)
1
>>> asyncio.run(work())
Start
End
```

```
>>> def process():
...     logger.info("Message sent from the child")
...     logger.complete()
...
>>> logger.add(sys.stderr, enqueue=True)
1
>>> process = multiprocessing.Process(target=process)
>>> process.start()
>>> process.join()
Message sent from the child
```

catch(*exception=<class 'Exception'>*, *, *level='ERROR'*, *reraise=False*, *onerror=None*, *exclude=None*, *default=None*, *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 `Thread.run()` methods while using threads to propagate errors to the main logger thread.

Note that the visibility of variables values (which uses the great `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.

- **onerror** (`callable`, optional) – A function that will be called if an error occurs, once the message has been logged. It should accept the exception instance as it sole argument.

- **exclude** (`Exception`, optional) – A type of exception (or a tuple of types) that will be purposely ignored and hence propagated to the caller without being logged.

- **default** (`Any`, optional) – The value to be returned by the decorated function if an error occurred without being re-raised.

- **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()
    └ <function g at 0x7f225fe2bc80>
> File "program.py", line 10, in g
    f(0)
    └ <function f at 0x7f225fe2b9d8>
  File "program.py", line 6, in f
    100 / x
          └ 0
ZeroDivisionError: division by zero
```

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

```
>>> # Use 'onerror' to prevent the program exit code to be 0 (if 'reraise=False')
→while
>>> # also avoiding the stacktrace to be duplicated on stderr (if 'reraise=True
→').
>>> @logger.catch(onerror=lambda _: sys.exit(1))
```

```
... def main():
...       1 / 0
```

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

Parametrize a logging call to slightly change generated log message.

Note that it's not possible to chain *opt()* calls, the last one takes precedence over the others as it will "reset" the options to their default values.

>   **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.
>
>   - **colors** (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.
>
>   - **capture** (bool, optional) – If False, the **kwargs of logged message will not automatically populate the extra dict (although they are still used for formatting).
>
>   - **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.
>
>   - **ansi** (bool, optional) – Deprecated since version 0.4.1: the ansi parameter will be removed in Loguru 1.0.0, it is replaced by colors which is a more appropriate name.
>
>   **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:␣
↪263130836933693530167218012160000000
```

```
>>> logger.opt(colors=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
```

```
>>> logger.opt(capture=False).info("Displayed but not captured: {value}",␣
↪value=123)
[18:11:41] Displayed but not captured: 123
```

```
>>> 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}")
>>> 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
```

contextualize(***kwargs*)

> Bind attributes to the context-local `extra` dict while inside the `with` block.
>
> Contrary to *bind()* there is no `logger` returned, the `extra` dict is modified in-place and updated globally. Most importantly, it uses `contextvars` which means that contextualized values are unique to each threads and asynchronous tasks.
>
> The `extra` dict will retrieve its initial state once the context manager is exited.
>
> > **Parameters**
> > ****kwargs** – Mapping between keys and values that will be added to the context-local `extra` dict.
> >
> > **Returns**
> > context manager / decorator – A context manager (usable as a decorator too) that will bind the attributes once entered and restore the initial state of the `extra` dict while exited.
>
> ### Examples
>
> ```
> >>> logger.add(sys.stderr, format="{message} | {extra}")
> 1
> >>> def task():
> ...     logger.info("Processing!")
> ...
> >>> with logger.contextualize(task_id=123):
> ...     task()
> ...
> Processing! | {'task_id': 123}
> >>> logger.info("Done.")
> Done. | {}
> ```

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.
>
> The logger can be patched multiple times. In this case, the functions are called in the same order as they are added.
>
> > **Parameters**
> > **patcher** (*callable*) – 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` tag 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, its `name` and its `no` are required. A `color` and an `icon` can also be specified or will be empty by default.

To update an existing level, pass its `name` with the parameters to be changed. It is not possible to modify the `no` of a level once it has been added.

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.

Raises

> `ValueError` – If there is no level registered with such `name`.

Examples

```
>>> level = logger.level("ERROR")
>>> print(level)
Level(name='ERROR', no=40, color='<red><bold>', icon='')
>>> logger.add(sys.stderr, format="{level.no} {level.icon} {message}")
1
>>> logger.level("CUSTOM", no=15, color="<blue>", icon="@")
Level(name='CUSTOM', no=15, color='<blue>', icon='@')
>>> logger.log("CUSTOM", "Logging...")
15 @ Logging...
>>> logger.level("WARNING", icon=r"/!\")
Level(name='WARNING', no=30, color='<yellow><bold>', icon='/!\\')
>>> 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 equal 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*, *patcher=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.

To load the configuration directly from a file, such as JSON or YAML, it is also possible to use the `loguru-config` library developed by @erezinman.

>**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.
>
>- **patcher** (`callable`, 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 `patcher` 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="¤", color="")],
...     extra={"common_to_all": "default"},
...     patcher=lambda record: record["extra"].update(some_value=42),
...     activation=[("my_module.secret", False), ("another_library.module",␣
```

(continues on next page)

```
→True)],
... )
[1, 2]
```

```
>>> # Set a default "extra" dict to logger across all modules, without "bind()"
>>> extra = {"context": "foo"}
>>> logger.configure(extra=extra)
>>> logger.add(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** (`callable` 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(*_Logger__message*, **args*, ***kwargs*)

> Log message.format(*args, **kwargs) with severity `'TRACE'`.

debug(*_Logger__message*, **args*, ***kwargs*)

> Log message.format(*args, **kwargs) with severity `'DEBUG'`.

info(*_Logger__message*, **args*, ***kwargs*)

> Log message.format(*args, **kwargs) with severity `'INFO'`.

success(*_Logger__message*, **args*, ***kwargs*)

> Log message.format(*args, **kwargs) with severity `'SUCCESS'`.

warning(*_Logger__message*, **args*, ***kwargs*)

> Log message.format(*args, **kwargs) with severity `'WARNING'`.

error(*_Logger__message*, **args*, ***kwargs*)

> Log message.format(*args, **kwargs) with severity `'ERROR'`.

critical(*_Logger__message*, **args*, ***kwargs*)

> Log message.format(*args, **kwargs) with severity `'CRITICAL'`.

exception(*_Logger__message*, **args*, ***kwargs*)

> Convenience method for logging an `'ERROR'` with exception information.

log(*_Logger__level*, *_Logger__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.

2.2 Type hints

Loguru relies on a stub file to document its types. This implies that these types are not accessible during execution of your program, however they can be used by type checkers and IDE. Also, this means that your Python interpreter has to support postponed evaluation of annotations to prevent error at runtime. This is achieved with a `__future__` import in Python 3.7+ or by using string literals for earlier versions.

A basic usage example could look like this:

```python
from __future__ import annotations

import loguru
from loguru import logger

def good_sink(message: loguru.Message):
    print("My name is", message.record["name"])

def bad_filter(record: loguru.Record):
    return record["invalid"]

logger.add(good_sink, filter=bad_filter)
```

```
$ mypy test.py
test.py:8: error: TypedDict "Record" has no key 'invalid'
Found 1 error in 1 file (checked 1 source file)
```

There are several internal types to which you can be exposed using Loguru's public API, they are listed here and might be useful to type hint your code:

- `Logger`: the usual *Logger* object (also returned by *opt()*, *bind()* and *patch()*).
- `Message`: the formatted logging message sent to the sinks (a `str` with `record` attribute).
- `Record`: the `dict` containing all contextual information of the logged message.
- `Level`: the `namedtuple` returned by *level()* (with `name`, `no`, `color` and `icon` attributes).
- `Catcher`: the context decorator returned by *catch()*.
- `Contextualizer`: the context decorator returned by *contextualize()*.
- `AwaitableCompleter`: the awaitable object returned by *complete()*.
- `RecordFile`: the `record["file"]` with `name` and `path` attributes.
- `RecordLevel`: the `record["level"]` with `name`, `no` and `icon` attributes.
- `RecordThread`: the `record["thread"]` with `id` and `name` attributes.
- `RecordProcess`: the `record["process"]` with `id` and `name` attributes.
- `RecordException`: the `record["exception"]` with `type`, `value` and `traceback` attributes.

If that is not enough, one can also use the `loguru-mypy` library developed by @kornicameister. Plugin can be installed separately using:

```
pip install loguru-mypy
```

It helps to catch several possible runtime errors by performing additional checks like:

- `opt(lazy=True)` loggers accepting only `typing.Callable[[], typing.Any]` arguments

- `opt(record=True)` loggers wrongly calling log handler like so `logger.info(..., record={})`

- and even more...

For more details, go to official documentation of `loguru-mypy`.

See also: type-hints-source.

- *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()*
    - *complete()*
    - *catch()*
    - *opt()*
    - *bind()*
    - *contextualize()*
    - *patch()*
    - *level()*
    - *disable()*
    - *enable()*
    - *configure()*
    - *parse()*
    - *trace()*
    - *debug()*
    - *info()*
    - *success()*
    - *warning()*
    - *error()*
    - *critical()*
    - *log()*
    - *exception()*
- *Type hints*

HELP & GUIDES

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 or 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`. Another major benefit is that Python resolves encoding issues on `sys.stderr` by escaping faulty characters (`"backslashreplace"` policy) while it raises an `UnicodeEncodeError` (`"strict"` policy) on `sys.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 sink `level` and `filter` parameters. The `propagate` attribute and `disable()` function 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:

```python
def specific_only(record):
    return "specific" in record["extra"]


logger.add("specific.log", filter=specific_only)


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:

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

```python
logger.setLevel(logging.DEBUG)


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


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:

```python
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:

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

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

```python
# 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:

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

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

```python
logger.getLevelName(33)   # => "CUSTOM"
logger.level("CUSTOM")    # => (name='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 Replacing `captureWarnings()` function

The `captureWarnings()` function which redirects alerts from the `warnings` module to the logging system can be implemented by simply replacing `warnings.showwarning()` function as follow:

```python
import warnings
from loguru import logger


showwarning_ = warnings.showwarning

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


warnings.showwarning = showwarning
```

3.1.12 Replacing `assertLogs()` method from `unittest` library

The `assertLogs()` method defined in the `unittest` from standard library is used to capture and test logged messages. However, it can't be made compatible with Loguru. It needs to be replaced with a custom context manager possibly implemented as follows:

```python
from contextlib import contextmanager


@contextmanager
def capture_logs(level="INFO", format="{level}:{name}:{message}"):
    """Capture loguru-based logs."""
    output = []
    handler_id = logger.add(output.append, level=level, format=format)
    yield output
    logger.remove(handler_id)
```

It provides the list of logged messages for each of which you can access the `record` attribute.

3.1.13 Replacing `caplog` fixture from `pytest` library

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

```python
from loguru import logger


def some_func(a, b):
    if a < 0:
        logger.warning("Oh no!")
    return a + b


def test_some_func(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 the caplog handler. This is done by overriding the `caplog` fixture to capture its handler. In your `conftest.py` file, add the following:

```python
import pytest
from loguru import logger
from _pytest.logging import LogCaptureFixture

@pytest.fixture
def caplog(caplog: LogCaptureFixture):
    handler_id = logger.add(
        caplog.handler,
        format="{message}",
        level=0,
        filter=lambda record: record["level"].no >= caplog.handler.level,
        enqueue=False,  # Set to 'True' if your test is spawning child processes.
    )
    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 and GH#474. You can also install and use the `pytest-loguru` package created by @mcarans.

Note that if you want Loguru logs to be propagated to Pytest terminal reporter, you can do so by overriding the `reportlog` fixture as follows:

```python
import pytest
from loguru import logger

@pytest.fixture
def reportlog(pytestconfig):
    logging_plugin = pytestconfig.pluginmanager.getplugin("logging-plugin")
    handler_id = logger.add(logging_plugin.report_handler, format="{message}")
    yield
    logger.remove(handler_id)
```

3.2 Code snippets and recipes for `loguru`

3.2.1 Security considerations when using Loguru

Firstly, if you use `pickle` to load log messages (e.g. from the network), make sure the source is trustable or sign the data to verify its authenticity before deserializing it. If you do not take these precautions, malicious code could be executed by an attacker. You can read more details in this article: What's so dangerous about pickles?

```python
import hashlib
import hmac
import pickle

def client(connection):
    data = pickle.dumps("Log message")
    digest =  hmac.digest(b"secret-shared-key", data, hashlib.sha1)
    connection.send(digest + b" " + data)
```

```python
def server(connection):
    expected_digest, data = connection.read().split(b" ", 1)
    data_digest = hmac.digest(b"secret-shared-key", data, hashlib.sha1)
    if not hmac.compare_digest(data_digest, expected_digest):
        print("Integrity error")
    else:
        message = pickle.loads(data)
        logger.info(message)
```

You should also avoid logging a message that could be maliciously hand-crafted by an attacker. Calling `logger.debug(message, value)` is roughly equivalent to calling `print(message.format(value))` and the same safety rules apply. In particular, an attacker could force printing of assumed hidden variables of your application. Here is an article explaining the possible vulnerability: Be Careful with Python's New-Style String Format.

```python
SECRET_KEY = 'Y0UC4NTS33Th1S!'


class SomeValue:
    def __init__(self, value):
        self.value = value


# If user types "{value.__init__.__globals__[SECRET_KEY]}" then the secret key is␣
↪displayed.
message = "[Custom message] " + input()
logger.info(message, value=SomeValue(10))
```

Another danger due to external input is the possibility of a log injection attack. Consider that you may need to escape user values before logging them: Is your Python code vulnerable to log injection?

```python
logger.add("file.log", format="{level} {message}")

# If value is "Josh logged in.\nINFO User James" then there will appear to be two log␣
↪entries.
username = external_data()
logger.info("User " + username + " logged in.")
```

Note that by default, Loguru will display the value of existing variables when an `Exception` is logged. This is very useful for debugging but could lead to credentials appearing in log files. Make sure to turn it off in production (or set the `LOGURU_DIAGNOSE=NO` environment variable).

```python
logger.add("out.log", diagnose=False)
```

Another thing you should consider is to change the access permissions of your log file. Loguru creates files using the built-in `open()` function, which means by default they might be read by a different user than the owner. If this is not desirable, be sure to modify the default access rights.

```python
def opener(file, flags):
    return os.open(file, flags, 0o600)


logger.add("combined.log", opener=opener)
```

3.2.2 Avoiding logs to be printed twice on the terminal

The logger is pre-configured for convenience with a default handler which writes messages to `sys.stderr`. You should *remove()* it first if you plan to *add()* another handler logging messages to the console, otherwise you may end up with duplicated logs.

```
logger.remove()  # Remove all handlers added so far, including the default one.
logger.add(sys.stderr, level="WARNING")
```

3.2.3 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, tough, 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)  # For the default handler, it's actually '0'.
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'
        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")
```

3.2.4 Configuring Loguru to be used by a library or an application

A clear distinction must be made between the use of Loguru within a library or an application.

In case of an application, you can add handlers from anywhere in your code. It's advised though to configure the logger from within a `if __name__ == "__main__":` block inside the entry file of your script.

However, if your work is intended to be used as a library, you usually should not add any handler. This is user responsibility to configure logging according to its preferences, and it's better not to interfere with that. Indeed, since Loguru is based on a single common logger, handlers added by a library will also receive user logs, which is generally not desirable.

By default, a third-library should not emit logs except if specifically requested. For this reason, there exist the *disable()* and *enable()* methods. Make sure to first call `logger.disable("mylib")`. This avoids library logs to be mixed with those of the user. The user can always call `logger.enable("mylib")` if he wants to access the logs of your library.

If you would like to ease logging configuration for your library users, it is advised to provide a function like `configure_logger()` in charge of adding the desired handlers. This will allow the user to activate the logging only if he needs to.

To summarize, let's look at this hypothetical package (none of the listed files are required, it all depends on how you plan your project to be used):

```
mypackage
├── __init__.py
├── __main__.py
├── main.py
└── mymodule.py
```

Files relate to Loguru as follows:

- File `__init__.py`:

  - It is the entry point when your project is used as a library (`import mypackage`).

  - It should contain `logger.disable("mypackage")` unconditionally at the top level.

  - It should not call `logger.add()` as it modifies handlers configuration.

- File `__main__.py`:

- It is the entry point when your project is used as an application (`python -m mypackage`).

- It can contain logging configuration unconditionally at the top level.

- File `main.py`:

  - It is the entry point when your project is used as a script (`python mypackage/main.py`).

  - It can contain logging configuration inside an `if __name__ == "__main__":` block.

- File `mymodule.py`:

  - It is an internal module used by your project.

  - It can use the `logger` simply by importing it.

  - It does not need to configure anything.

3.2.5 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.

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

from loguru import logger


class SocketHandler:

    def __init__(self, host, port):
        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")
```

```python
# server.py
import socketserver
import pickle
```

(continues on next page)

```python
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"].no, 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.

Another possibility is to use a third party library like zmq for example.

```python
# client.py
import zmq
from zmq.log.handlers import PUBHandler
from loguru import logger

socket = zmq.Context().socket(zmq.PUB)
socket.connect("tcp://127.0.0.1:12345")
handler = PUBHandler(socket)
logger.add(handler)

logger.info("Logging from client")
```

```python
# server.py
import sys
import zmq
from loguru import logger

socket = zmq.Context().socket(zmq.SUB)
socket.bind("tcp://127.0.0.1:12345")
socket.subscribe("")

logger.configure(handlers=[{"sink": sys.stderr, "format": "{message}"}])

while True:
    _, message = socket.recv_multipart()
    logger.info(message.decode("utf8").strip())
```

3.2.6 Resolving `UnicodeEncodeError` and other encoding issues

When you write a log message, the handler may need to encode the received unicode string to a specific sequence of bytes. The `encoding` used to perform this operation varies depending on the sink type and your environment. Problem may occur if you try to write a character which is not supported by the handler `encoding`. In such case, it's likely that Python will raise an `UnicodeEncodeError`.

For example, this may happen while printing to the terminal:

```
print("天")
# UnicodeEncodeError: 'charmap' codec can't encode character '\u5929' in position 0:␣
↪character maps to <undefined>
```

A similar error may occur while writing to a file which has not been opened using an appropriate encoding. Most common problem happen while logging to standard output or to a file on Windows. So, how to avoid such error? Simply by properly configuring your handler so that it can process any kind of unicode string.

If you are encountering this error while logging to `stdout`, you have several options:

- Use `sys.stderr` instead of `sys.stdout` (the former will escape faulty characters rather than raising exception)
- Set the `PYTHONIOENCODING` environment variable to `utf-8`
- Call `sys.stdout.reconfigure()` with `encoding='utf-8'` and / or `errors='backslashreplace'`

If you are using a file sink, you can configure the `errors` or `encoding` parameter while adding the handler like `logger.add("file.log", encoding="utf8")` for example. All additional `**kwargs` argument are passed to the built-in `open()` function.

For other types of handlers, you have to check if there is a way to parametrize encoding or fallback policy.

3.2.7 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:

```python
import functools
from loguru import logger


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

    def wrapper(func):
        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:

```python
@logger_wraps()
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:

```python
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
```

3.2.8 Using logging function based on custom added levels

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

```python
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:

```python
from functools import partialmethod

logger.__class__.foobar = partialmethod(logger.__class__.log, "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`. 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).

3.2.9 Setting permissions on created log files

To set desired permissions on created log files, use the `opener` argument to pass in a custom opener with permissions octal:

```python
def opener(file, flags):
    return os.open(file, flags, 0o600)  # read/write by owner only

logger.add("foo.log", rotation="100 kB", opener=opener)
```

When using an opener argument, all created log files including ones created during rotation will use the initially provided opener.

Note that the provided mode will be masked out by the OS umask value (describing which bits are *not* to be set when creating a file or directory). This value is conventionally equals to `0o022`, which means specifying a `0o666` mode will result in a `0o666 - 0o022 = 0o644` file permission in this case (which is actually the default). It is possible to change the umask value by first calling `os.umask()`, but this needs to be done with careful consideration, as it changes the value globally and can cause security issues.

3.2.10 Preserving an `opt()` parameter for the whole module

Supposing you wish to color each of your log messages without having to call `logger.opt(colors=True)` every time, you can add this at the very beginning of your module:

```python
logger = logger.opt(colors=True)

logger.info("It <green>works</>!")
```

However, it should be noted that it's not possible to chain *opt()* calls, using this method again will reset the `colors` option to its default value (which is `False`). For this reason, it is also necessary to patch the *opt()* method so that all subsequent calls continue to use the desired value:

```python
from functools import partial

logger = logger.opt(colors=True)
logger.opt = partial(logger.opt, colors=True)

logger.opt(raw=True).info("It <green>still</> works!\n")
```

3.2.11 Serializing log messages using a custom function

Each handler added with `serialize=True` will create messages by converting the logging record to a valid JSON string. Depending on the sink for which the messages are intended, it may be useful to make changes to the generated string. Instead of using the `serialize` parameter, you can implement your own serialization function and use it directly in your sink:

```python
def serialize(record):
    subset = {"timestamp": record["time"].timestamp(), "message": record["message"]}
    return json.dumps(subset)

def sink(message):
    serialized = serialize(message.record)
    print(serialized)

logger.add(sink)
```

If you need to send structured logs to a file (or any kind of sink in general), a similar result can be obtained by using a custom `format` function:

```python
def formatter(record):
    # Note this function returns the string to be formatted, not the actual message to
    →be logged
    record["extra"]["serialized"] = serialize(record)
    return "{extra[serialized]}\n"

logger.add("file.log", format=formatter)
```

You can also use *patch()* for this, so the serialization function will be called only once in case you want to use it in multiple sinks:

```python
def patching(record):
    record["extra"]["serialized"] = serialize(record)

logger = logger.patch(patching)

# Note that if "format" is not a function, possible exception will be appended to the
→message
logger.add(sys.stderr, format="{extra[serialized]}")
logger.add("file.log", format="{extra[serialized]}")
```

3.2.12 Rotating log file based on both size and time

The `rotation` argument of file sinks accept size or time limits but not both for simplification reasons. However, it is possible to create a custom function to support more advanced scenarios:

```python
import datetime

class Rotator:
    def __init__(self, *, size, at):
        now = datetime.datetime.now()

        self._size_limit = size
```

```python
        self._time_limit = now.replace(hour=at.hour, minute=at.minute, second=at.second)

        if now >= self._time_limit:
            # The current time is already past the target time so it would rotate␣
→already.
            # Add one day to prevent an immediate rotation.
            self._time_limit += datetime.timedelta(days=1)

    def should_rotate(self, message, file):
        file.seek(0, 2)
        if file.tell() + len(message) > self._size_limit:
            return True
        excess = message.record["time"].timestamp() - self._time_limit.timestamp()
        if excess >= 0:
            elapsed_days = datetime.timedelta(seconds=excess).days
            self._time_limit += datetime.timedelta(days=elapsed_days + 1)
            return True
        return False


# Rotate file if over 500 MB or at midnight every day
rotator = Rotator(size=5e+8, at=datetime.time(0, 0, 0))
logger.add("file.log", rotation=rotator.should_rotate)
```

3.2.13 Adapting colors and format of logged messages dynamically

It is possible to customize the colors of your logs thanks to several *markup tags*. Those are used to configure the `format` of your handler. By creating a appropriate formatting function, you can easily define colors depending on the logged message.

For example, if you want to associate each module with a unique color:

```python
from collections import defaultdict
from random import choice

colors = ["blue", "cyan", "green", "magenta", "red", "yellow"]
color_per_module = defaultdict(lambda: choice(colors))

def formatter(record):
    color_tag = color_per_module[record["name"]]
    return "<" + color_tag + ">[{name}]</> <bold>{message}</>\n{exception}"

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

If you need to dynamically colorize the `record["message"]`, make sure that the color tags appear in the returned format instead of modifying the message:

```python
def rainbow(text):
    colors = ["red", "yellow", "green", "cyan", "blue", "magenta"]
    chars = ("<{}>{}</>".format(colors[i % len(colors)], c) for i, c in enumerate(text))
    return "".join(chars)

def formatter(record):
```

```
    rainbow_message = rainbow(record["message"])
    # Prevent '{}' in message (if any) to be incorrectly parsed during formatting
    escaped = rainbow_message.replace("{", "{{").replace("}", "}}")
    return "<b>{time}</> " + escaped + "\n{exception}"

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

3.2.14 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. For this purpose, you can use Python's string formatting directives, like in this example:

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

Here, <, ^ and > will left, center, and right-align the respective keys, and pad them to a maximum length.

Other 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)
```

3.2.15 Customizing the formatting of exceptions

Loguru will automatically add the traceback of occurring exception while using logger.exception() or logger.opt(exception=True):

```
def inverse(x):
    try:
        1 / x
    except ZeroDivisionError:
        logger.exception("Oups...")
```

```
if __name__ == "__main__":
    inverse(0)
```

```
2019-11-15 10:01:13.703 | ERROR    | __main__:inverse:8 - Oups...
Traceback (most recent call last):
File "foo.py", line 6, in inverse
    1 / x
ZeroDivisionError: division by zero
```

If the handler is added with `backtrace=True`, the traceback is extended to see where the exception came from:

```
2019-11-15 10:11:32.829 | ERROR    | __main__:inverse:8 - Oups...
Traceback (most recent call last):
  File "foo.py", line 16, in <module>
    inverse(0)
> File "foo.py", line 6, in inverse
    1 / x
ZeroDivisionError: division by zero
```

If the handler is added with `diagnose=True`, then the traceback is annotated to see what caused the problem:

```
Traceback (most recent call last):

File "foo.py", line 6, in inverse
    1 / x
        └ 0

ZeroDivisionError: division by zero
```

It is possible to further personalize the formatting of exception by adding an handler with a custom `format` function. For example, supposing you want to format errors using the `stackprinter` library:

```python
import stackprinter

def format(record):
    format_ = "{time} {message}\n"

    if record["exception"] is not None:
        record["extra"]["stack"] = stackprinter.format(record["exception"])
        format_ += "{extra[stack]}\n"

    return format_

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

```
2019-11-15T10:46:18.059964+0100 Oups...
File foo.py, line 17, in inverse
    15   def inverse(x):
    16       try:
--> 17           1 / x
    18       except ZeroDivisionError:
```

```
    ..............................................
    x = 0
    ..............................................

ZeroDivisionError: division by zero
```

3.2.16 Displaying a stacktrace without using the error context

It may be useful in some cases to display the traceback at the time your message is logged, while no exceptions have been raised. Although this feature is not built-in into Loguru as it is more related to debugging than logging, it is possible to *patch()* your logger and then display the stacktrace as needed (using the `traceback` module):

```python
import traceback

def add_traceback(record):
    extra = record["extra"]
    if extra.get("with_traceback", False):
        extra["traceback"] = "\n" + "".join(traceback.format_stack())
    else:
        extra["traceback"] = ""

logger = logger.patch(add_traceback)
logger.add(sys.stderr, format="{time} - {message}{extra[traceback]}")

logger.info("No traceback")
logger.bind(with_traceback=True).info("With traceback")
```

Here is another example that demonstrates how to prefix the logged message with the full call stack:

```python
import traceback
from itertools import takewhile

def tracing_formatter(record):
    # Filter out frames coming from Loguru internals
    frames = takewhile(lambda f: "/loguru/" not in f.filename, traceback.extract_stack())
    stack = " > ".join("{}:{}:{}".format(f.filename, f.name, f.lineno) for f in frames)
    record["extra"]["stack"] = stack
    return "{level} | {extra[stack]} - {message}\n{exception}"

def foo():
    logger.info("Deep call")

def bar():
    foo()

logger.remove()
logger.add(sys.stderr, format=tracing_formatter)

bar()
# Output: "INFO | script.py:<module>:23 > script.py:bar:18 > script.py:foo:15 - Deep call
↪"
```

3.2.17 Manipulating newline terminator to write multiple logs on the same line

You can temporarily log a message on a continuous line by combining the use of *bind()*, *opt()* and a custom `format` function. This is especially useful if you want to illustrate a step-by-step process in progress, for example:

```python
def formatter(record):
    end = record["extra"].get("end", "\n")
    return "[{time}] {message}" + end + "{exception}"

logger.add(sys.stderr, format=formatter)
logger.add("foo.log", mode="w")

logger.bind(end="").debug("Progress: ")

for _ in range(5):
    logger.opt(raw=True).debug(".")

logger.opt(raw=True).debug("\n")

logger.info("Done")
```

```
[2020-03-26T22:47:01.708016+0100] Progress: .....
[2020-03-26T22:47:01.709031+0100] Done
```

Note, however, that you may encounter difficulties depending on the sinks you use. Logging is not always appropriate for this type of end-user message.

3.2.18 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 using `contextlib.redirect_stdout()`. 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):

```python
import contextlib
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.opt(depth=1).log(self._level, line.rstrip())

    def flush(self):
        pass

logger.remove()
```

(continues on next page)

```
logger.add(sys.__stdout__)

stream = StreamToLogger()
with contextlib.redirect_stdout(stream):
    print("Standard output is sent to added handlers.")
```

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
```

Alternatively, if you want to emit warnings based on logged messages, you can simply use `warnings.warn()` as a sink:

```
logger.add(warnings.warn, format="{message}", filter=lambda record: record["level"].name
→== "WARNING")
```

3.2.19 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.

3.2.20 Interoperability with `tqdm` iterations

Trying to use the Loguru's `logger` during an iteration wrapped by the `tqdm` library may disturb the displayed progress bar. As a workaround, one can use the `tqdm.write()` function instead of writings logs directly to `sys.stderr`:

```
import time

from loguru import logger
from tqdm import tqdm
```

```
logger.remove()
logger.add(lambda msg: tqdm.write(msg, end=""), colorize=True)

logger.info("Initializing")

for x in tqdm(range(100)):
    logger.info("Iterating #{}", x)
    time.sleep(0.1)
```

You may encounter problems with colorization of your logs after importing `tqdm` using Spyder on Windows. This issue is discussed in GH#132. You can easily circumvent the problem by calling `colorama.deinit()` right after your import.

3.2.21 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.

3.2.22 Creating independent loggers with separate set of handlers

Loguru is fundamentally designed to be usable with exactly one global `logger` object dispatching logging messages to the configured handlers. In some circumstances, it may be useful to have specific messages logged to specific handlers.

For example, supposing you want to split your logs in two files based on an arbitrary identifier, you can achieve that by combining *bind()* and `filter`:

```
from loguru import logger

def task_A():
    logger_a = logger.bind(task="A")
    logger_a.info("Starting task A")
    do_something()
    logger_a.success("End of task A")

def task_B():
    logger_b = logger.bind(task="B")
```

```
    logger_b.info("Starting task B")
    do_something_else()
    logger_b.success("End of task B")


logger.add("file_A.log", filter=lambda record: record["extra"]["task"] == "A")
logger.add("file_B.log", filter=lambda record: record["extra"]["task"] == "B")


task_A()
task_B()
```

That way, `"file_A.log"` and `"file_B.log"` will only contains logs from respectively the `task_A()` and `task_B()` function.

Now, supposing that you have a lot of these tasks. It may be a bit cumbersome to configure every handlers like this. Most importantly, it may unnecessarily slow down your application as each log will need to be checked by the `filter` function of each handler. In such case, it is recommended to rely on the `copy.deepcopy()` built-in method that will create an independent `logger` object. If you add a handler to a deep copied `logger`, it will not be shared with others functions using the original `logger`:

```python
import copy
from loguru import logger


def task(task_id, logger):
    logger.info("Starting task {}", task_id)
    do_something(task_id)
    logger.success("End of task {}", task_id)


logger.remove()


for task_id in ["A", "B", "C", "D", "E"]:
    logger_ = copy.deepcopy(logger)
    logger_.add("file_%s.log" % task_id)
    task(task_id, logger_)
```

Note that you may encounter errors if you try to copy a `logger` to which non-picklable handlers have been added. For this reason, it is generally advised to remove all handlers before calling `copy.deepcopy(logger)`.

3.2.23 Compatibility with `multiprocessing` using `enqueue` argument

On Linux, thanks to `os.fork()` there is no pitfall while using the `logger` inside another process started by the `multiprocessing` module. The child process will automatically inherit added handlers, the `enqueue=True` parameter is optional but is recommended as it would avoid concurrent access of your sink:

```python
# Linux implementation
import multiprocessing
from loguru import logger


def my_process():
    logger.info("Executing function in child process")
    logger.complete()


if __name__ == "__main__":
```

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

    process = multiprocessing.Process(target=my_process)
    process.start()
    process.join()

    logger.info("Done")
```

Things get a little more complicated on Windows. Indeed, this operating system does not support forking, so Python has to use an alternative method to create sub-processes called "spawning". This procedure requires the whole file where the child process is created to be reloaded from scratch. This does not interoperate very well with Loguru, causing handlers to be added twice without any synchronization or, on the contrary, not being added at all (depending on `add()` and `remove()` being called inside or outside the `__main__` branch). For this reason, the `logger` object need to be explicitly passed as an initializer argument of your child process:

```python
# Windows implementation
import multiprocessing
from loguru import logger


def my_process(logger_):
    logger_.info("Executing function in child process")
    logger_.complete()


if __name__ == "__main__":
    logger.remove()  # Default "sys.stderr" sink is not picklable
    logger.add("file.log", enqueue=True)

    process = multiprocessing.Process(target=my_process, args=(logger, ))
    process.start()
    process.join()

    logger.info("Done")
```

Windows requires the added sinks to be picklable or otherwise will raise an error while creating the child process. Many stream objects like standard output and file descriptors are not picklable. In such case, the `enqueue=True` argument is required as it will allow the child process to only inherit the queue object where logs are sent.

The `multiprocessing` library is also commonly used to start a pool of workers using for example `map()` or `apply()`. Again, it will work flawlessly on Linux, but it will require some tinkering on Windows. You will probably not be able to pass the `logger` as an argument for your worker functions because it needs to be picklable, but although handlers added using `enqueue=True` are "inheritable", they are not "picklable". Instead, you will need to make use of the `initializer` and `initargs` parameters while creating the `Pool` object in a way allowing your workers to access the shared `logger`. You can either assign it to a class attribute or override the global logger of your child processes:

```python
# workers_a.py
class Worker:

    _logger = None

    @staticmethod
    def set_logger(logger_):
        Worker._logger = logger_
```

```
    def work(self, x):
        self._logger.info("Square rooting {}", x)
        return x**0.5
```

```python
# workers_b.py
from loguru import logger

def set_logger(logger_):
    global logger
    logger = logger_

def work(x):
    logger.info("Square rooting {}", x)
    return x**0.5
```

```python
# main.py
from multiprocessing import Pool
from loguru import logger
import workers_a
import workers_b

if __name__ == "__main__":
    logger.remove()
    logger.add("file.log", enqueue=True)

    worker = workers_a.Worker()
    with Pool(4, initializer=worker.set_logger, initargs=(logger, )) as pool:
        results = pool.map(worker.work, [1, 10, 100])

    with Pool(4, initializer=workers_b.set_logger, initargs=(logger, )) as pool:
        results = pool.map(workers_b.work, [1, 10, 100])

    logger.info("Done")
```

Independently of the operating system, note that the process in which a handler is added with `enqueue=True` is in charge of the queue internally used. This means that you should avoid to `.remove()` such handler from the parent process is any child is likely to continue using it. More importantly, note that a `Thread` is started internally to consume the queue. Therefore, it is recommended to call *complete()* before leaving `Process` to make sure the queue is left in a stable state.

Another thing to keep in mind when dealing with multiprocessing is the fact that handlers created with `enqueue=True` create a queue internally in the default multiprocessing context. If they are passed through to a subprocesses instantiated within a different context (e.g. with `multiprocessing.get_context("spawn")` on linux, where the default context is `"fork"`) it will most likely result in crashing the subprocess. This is also noted in the python multiprocessing docs. To prevent any problems, you should specify the context to be used by Loguru while adding the handler. This can be done by passing the `context` argument to the `add()` method:

```python
import multiprocessing
from loguru import logger
import workers_a

if __name__ == "__main__":
```

```python
    context = multiprocessing.get_context("spawn")

    logger.remove()
    logger.add("file.log", enqueue=True, context=context)

    worker = workers_a.Worker()
    with context.Pool(4, initializer=worker.set_logger, initargs=(logger, )) as pool:
        results = pool.map(worker.work, [1, 10, 100])
```

PROJECT INFORMATION

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).

Prefer using a relatively recent Python version as some dependencies required for development may have dropped support for oldest Python versions. 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 venv env
$ source env/bin/activate
```

4. Install *Loguru* in development mode:

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

5. Install pre-commit hooks that will check your commits:

```
$ pre-commit install --install-hooks
```

6. Create a new branch from `master`:

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

7. Implement the modifications wished. During the process of development, honor PEP 8 as much as possible.

8. Add unit tests (don't hesitate to be exhaustive!) and ensure none are failing using:

```
$ tox -e tests
```

9. Remember to update documentation if required.

10. If your development modifies *Loguru* behavior, update the `CHANGELOG.rst` file with what you improved.

11. `add` and `commit` your changes, then `push` your local project:

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

12. If previous step failed due to the pre-commit hooks, fix reported errors and try again.

13. 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 SOFT-
WARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

4.3 Changelog

4.3.1 0.7.2 (2023-09-11)

- Add support for formatting of `ExceptionGroup` errors ([#805](#)).

- Fix possible `RuntimeError` when using `multiprocessing.set_start_method()` after importing the `logger` ([#974](#))

- Fix formatting of possible `__notes__` attached to an `Exception` ([#980](#)).

4.3.2 0.7.1 (2023-09-04)

- Add a new `context` optional argument to `logger.add()` specifying `multiprocessing` context (like `"spawn"` or `"fork"`) to be used internally instead of the default one ([#851](#)).

- Add support for true colors on Windows using ANSI/VT console when available ([#934](#), thanks [@tunaflsh](#)).

- Fix possible deadlock when calling `logger.complete()` with concurrent logging of an asynchronous sink ([#906](#)).

- Fix file possibly rotating too early or too late when re-starting an application around midnight ([#894](#)).

- Fix inverted `"<hide>"` and `"<strike>"` color tags ([#943](#), thanks [@tunaflsh](#)).

- Fix possible untraceable errors raised when logging non-unpicklable `Exception` instances while using `enqueue=True` ([#329](#)).

- Fix possible errors raised when logging non-picklable `Exception` instances while using `enqueue=True` ([#342](#), thanks [@ncoudene](#)).

- Fix missing seconds and microseconds when formatting timezone offset that requires such accuracy ([#961](#)).

- Raise `ValueError` if an attempt to use nanosecond precision for time formatting is detected ([#855](#)).

4.3.3 0.7.0 (2023-04-10)

- Update `InterceptHandler` recipe to make it compatible with Python 3.11 ([#654](#)).

- Add a new `watch` optional argument to file sinks in order to automatically re-create possibly deleted or changed file ([#471](#)).

- Make `patch()` calls cumulative instead of overriding the possibly existing patching function ([#462](#)).

- Make sinks added with `enqueue=True` and `catch=False` still process logged messages in case of internal exception ([#833](#)).

- Avoid possible deadlocks caused by re-using the logger inside a sink, a signal handler or a `__del__` method. Since the logger is not re-entrant, such misuse will be detected and will now generate a `RuntimeError` ([#712](#), thanks [@jacksmith15](#)).

- Fix file sink rotation using an aware `datetime.time` for which the timezone was ignored ([#697](#)).

- Fix logs colorization not automatically enabled for Jupyter Notebook and Google Colab ([#494](#)).

- Fix logs colorization not automatically enabled for Github Actions and others CI platforms ([#604](#)).

- Fix `logger.complete()` possibly hanging forever when `enqueue=True` and `catch=False` if internal thread killed due to `Exception` raised by sink ([#647](#)).

- Fix incompatibility with `freezegun` library used to simulate time ([#600](#)).

- Raise exception if `logger.catch()` is used to wrap a class instead of a function to avoid unexpected behavior ([#623](#)).

4.3.4 0.6.0 (2022-01-29)

- Remove internal use of `pickle.loads()` to fix the (finally rejected) security vulnerability referenced as [CVE-2022-0329](#) ([#563](#)).

- Modify coroutine sink to make it discard log messages when `loop=None` and no event loop is running (due to internally using `asyncio.get_running_loop()` in place of `asyncio.get_event_loop()`).

- Remove the possibility to add a coroutine sink with `enqueue=True` if `loop=None` and no event loop is running.

- Change default encoding of file sink to be `utf8` instead of `locale.getpreferredencoding()` ([#339](#)).

- Prevent non-ascii characters to be escaped while logging JSON message with `serialize=True` ([#575](#), thanks [@ponponon](#)).

- Fix `flake8` errors and improve code readability ([#353](#), thanks [@AndrewYakimets](#)).

4.3.5 0.5.3 (2020-09-20)

- Fix child process possibly hanging at exit while combining `enqueue=True` with third party library like `uwsgi` ([#309](#), thanks [@dstlmrk](#)).

- Fix possible exception during formatting of non-string messages ([#331](#)).

4.3.6 0.5.2 (2020-09-06)

- Fix `AttributeError` within handlers using `serialize=True` when calling `logger.exception()` outside of the context of an exception ([#296](#)).

- Fix error while logging an exception containing a non-picklable `value` to a handler with `enqueue=True` ([#298](#)).

- Add support for async callable classes (with `__call__` method) used as sinks ([#294](#), thanks [@jessekrubin](#)).

4.3.7 0.5.1 (2020-06-12)

- Modify the way the `extra` dict is used by `LogRecord` in order to prevent possible `KeyError` with standard `logging` handlers ([#271](#)).

- Add a new `default` optional argument to `logger.catch()`, it should be the returned value by the decorated function in case an error occurred ([#272](#)).

- Fix `ValueError` when using `serialize=True` in combination with `logger.catch()` or `logger.opt(record=True)` due to circular reference of the `record` dict ([#286](#)).

4.3.8 0.5.0 (2020-05-17)

- Remove the possibility to modify the severity `no` of levels once they have been added in order to prevent surprising behavior (#209).

- Add better support for "structured logging" by automatically adding `**kwargs` to the `extra` dict besides using these arguments to format the message. This behavior can be disabled by setting the new `.opt(capture=False)` parameter (#2).

- Add a new `onerror` optional argument to `logger.catch()`, it should be a function which will be called when an exception occurs in order to customize error handling (#224).

- Add a new `exclude` optional argument to `logger.catch()`, is should be a type of exception to be purposefully ignored and propagated to the caller without being logged (#248).

- Modify `complete()` to make it callable from non-asynchronous functions, it can thus be used if `enqueue=True` to make sure all messages have been processed (#231).

- Fix possible deadlocks on Linux when `multiprocessing.Process()` collides with `enqueue=True` or `threading` (#231).

- Fix `compression` function not executable concurrently due to file renaming (to resolve conflicts) being performed after and not before it (#243).

- Fix the filter function listing files for `retention` being too restrictive, it now matches files based on the pattern `"basename(.*).ext(.*)"` (#229).

- Fix the impossibility to `remove()` a handler if an exception is raised while the sink' `stop()` function is called (#237).

- Fix file sink left in an unstable state if an exception occurred during `retention` or `compression` process (#238).

- Fix situation where changes made to `record["message"]` were unexpectedly ignored when `opt(colors=True)`, causing "out-of-date" `message` to be logged due to implementation details (#221).

- Fix possible exception if a stream having an `isatty()` method returning `True` but not being compatible with `colorama` is used on Windows (#249).

- Fix exceptions occurring in coroutine sinks never retrieved and hence causing warnings (#227).

4.3.9 0.4.1 (2020-01-19)

- Deprecate the `ansi` parameter of `.opt()` in favor of `colors` which is a name more appropriate.

- Prevent unrelated files and directories to be incorrectly collected thus causing errors during the `retention` process (#195, thanks @gazpachoking).

- Strip color markups contained in `record["message"]` when logging with `.opt(ansi=True)` instead of leaving them as is (#198).

- Ignore color markups contained in `*args` and `**kwargs` when logging with `.opt(ansi=True)`, leave them as is instead of trying to use them to colorize the message which could cause undesirable errors (#197).

4.3.10 0.4.0 (2019-12-02)

- Add support for coroutine functions used as sinks and add the new `logger.complete()` asynchronous method to `await` them (#171).

- Add a way to filter logs using one level per module in the form of a `dict` passed to the `filter` argument (#148).

- Add type hints to annotate the public methods using a `.pyi` stub file (#162).

- Add support for `copy.deepcopy()` of the `logger` allowing multiple independent loggers with separate set of handlers (#72).

- Add the possibility to convert `datetime` to UTC before formatting (in logs and filenames) by adding `"!UTC"` at the end of the time format specifier (#128).

- Add the level `name` as the first argument of namedtuple returned by the `.level()` method.

- Remove `class` objects from the list of supported sinks and restrict usage of `**kwargs` in `.add()` to file sink only. User is in charge of instantiating sink and wrapping additional keyword arguments if needed, before passing it to the `.add()` method.

- Rename the `logger.configure()` keyword argument `patch` to `patcher` so it better matches the signature of `logger.patch()`.

- Fix incompatibility with `multiprocessing` on Windows by entirely refactoring the internal structure of the `logger` so it can be inherited by child processes along with added handlers (#108).

- Fix `AttributeError` while using a file sink on some distributions (like Alpine Linux) missing the `os.getxattr` and `os.setxattr` functions (#158, thanks @joshgordon).

- Fix values wrongly displayed for keyword arguments during exception formatting with `diagnose=True` (#144).

- Fix logging messages wrongly chopped off at the end while using standard `logging.Handler` sinks with `.opt(raw=True)` (#136).

- Fix potential errors during rotation if destination file exists due to large resolution clock on Windows (#179).

- Fix an error using a `filter` function "by name" while receiving a log with `record["name"]` equals to `None`.

- Fix incorrect record displayed while handling errors (if `catch=True`) occurring because of non-picklable objects (if `enqueue=True`).

- Prevent hypothetical `ImportError` if a Python installation is missing the built-in `distutils` module (#118).

- Raise `TypeError` instead of `ValueError` when a `logger` method is called with argument of invalid type.

- Raise `ValueError` if the built-in `format()` and `filter()` functions are respectively used as `format` and `filter` arguments of the `add()` method. This helps the user to understand the problem, as such a mistake can quite easily occur (#177).

- Remove inheritance of some record dict attributes to `str` (for `"level"`, `"file"`, `"thread"` and `"process"`).

- Give a name to the worker thread used when `enqueue=True` (#174, thanks @t-mart).

4.3.11 0.3.2 (2019-07-21)

- Fix exception during import when executing Python with `-s` and `-S` flags causing `site.USER_SITE` to be missing ([#114](#)).

4.3.12 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.

4.3.13 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).

4.3.14 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.

4.3.15 0.2.4 (2018-12-26)

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

4.3.16 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).

4.3.17 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).

4.3.18 0.2.1 (2018-12-08)

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

4.3.19 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).

4.3.20 0.1.0 (2018-12-07)

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

4.3.21 0.0.1 (2017-09-04)

Initial release.

PYTHON MODULE INDEX