Overview

Installation

```
pip install loguru
```

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~~

Take the tour

Ready to use out of the box without boilerplate

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

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

```
from loguru import logger

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

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

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

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

One answer: the `add()` function.

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

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

Easier file logging with rotation / retention / compression

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

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

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

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

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

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

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")
```

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)
```

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>")
```

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()`.

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

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.

Structured logging as needed

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

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

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

```
logger.add("file.log", format="{extra[ip]} {extra[user]} {message}")
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()))
```

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")
```

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!")
```

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}")
```

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.

```
# 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.

Entirely compatible with standard logging

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

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

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

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

Personalizable defaults through environment variables

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

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

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

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)}
```

Exhaustive notifier

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

```
import notifiers

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

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

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

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

~~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.