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.

CHAPTER 1

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 an handler? How to setup logs formatting? How to filter messages? How to set level?

One answer: the `start()` function.

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

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

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

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

logger.start("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 logfile? Did you ever noticed that exceptions occuring 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 catched 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.start(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. If you want async logging or need to use the same sink through different multiprocesses, you just have to enqueue the messages.

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

1.3.8 Fully descriptive exceptions

Logging exceptions that occur in your code is important to track bugs, but it's quite useless if you don't know why it failed. *Loguru* help you identify problems by allowing the entire stack trace to be displayed, including variables values.

The code:

```
logger.start("output.log", backtrace=True)  # Set 'False' to avoid leaking sensible
↪data in prod

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, catch point marked):

  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
```

1.3.9 Structured logging as needed

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

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

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

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

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("Exception with an "INFO" level")
logger.opt(ansi=True).info("Per message <blue>colors</blue>")
logger.opt(record=True).info("Log record attributes (eg. {record[thread].id})")
```

```
logger.opt(raw=True).info("Bypass sink formatting\n")
logger.opt(depth=1).info("Use parent stack context (useful within wrapped functions)")
```

1.3.11 Customizable levels

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

```
new_level = logger.level("SNAKY", no=8, 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.start("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 libary, remember to never call `start()` but use `disable()` instead so logging functions become no-op. If an user want to see your library's logs, he can `enable()` it again.

```
# For scripts
my_logging_config = dict(
    handlers=[{'sink': sys.stdout, 'colorize': False, format="{time} - {message}"}],
    extra={"user": "someone"}
)
logger.configure(**my_logging_config)

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

1.3.14 Entirely compatible with standard logging

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

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

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

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

logger.start(PropagateHandler())
```

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

```python
class InterceptHandler(logging.Handler):
    def emit(self, record):
        logger_opt = logger.opt(depth=6, exception=record.exc_info)
        logger_opt.log(record.levelno, record.getMessage())

logging.getLogger(None).addHandler(InterceptHandler())
```

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 parser which helps dealing with logs and regexes.

```python
from loguru import parser

pattern = r"(?P<time>.*) - (?P<level>[0-9]+) - (?P<message>.*)"

for groups in parser.parse("file.log", pattern):
    groups = parser.cast(groups, time=time.strptime, level=int)
    print("Parsed message at {} with severity {}".format(groups["time"], groups["level
    →"]))
```

1.3.17 Exhaustive notifier

Receive an e-mail when your program fail unexpectedly or send many other kind of notifications using *Loguru* notifier.

```python
from loguru import notifier

gmail_notifier = notifier.gmail(to="dest@gmail.com", username="you@gmail.com",␣
→password="abc123")

# Send a notification
gmail_notifier.send("The application is running!")
```

```
# Be alerted on each error messages
logger.start(gmail_notifier.send, level="ERROR")
```

1.3.18 10x faster than built-in logging

Although logging impact on performances is in most cases negligeable, 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 pre-instanced objects to facilitate dealing with logging in Python.

Pick one: `from loguru import logger, notifier, parser`

2.1 loguru.logger

class Logger
> An object to dispatch logging messages to configured handlers.
>
> The *Logger* is the core objet 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.
>
> Handlers to which send log messages are added using the *start()* method. Note that you can use the *Logger* right after import as it comes pre-configured. Messages can be logged with different severity levels and using braces attributes like the `str.format()` method do.
>
> Once a message is logged, a "record" is associated with it. This record is a dict wich contains several 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.
>
> **start**(*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*, *enqueue=False*, *catch=True*, ***kwargs*)
> > Start sending log messages to a sink adequately configured.
> >
> > **Parameters**
> >
> > > - **sink** (`file-like object`, `str`, `pathlib.Path`, `function`, `logging.Handler` or `class`) – An object in charge of receiving formatted logging messages and propagating them to an appropriate endpoint.

- **level** (`int` or `str`, optional) – The minimum severity level from which logged messages should be send to the sink.

- **format** (`str` or `function`, optional) – The template used to format logged messages before being sent to the sink.

- **filter** (`function` or `str`, optional) – A directive used to optionally filter out logged messages before they are send to the sink.

- **colorize** (`bool`, optional) – Whether or not the color markups contained in the formatted message should be converted to ansi codes for terminal coloration, ore stripped otherwise. If `None`, the choice is automatically made based on the sink being a tty or not.

- **serialize** (`bool`, optional) – Whether or not the logged message and its records should be first converted to a JSON string before being sent to the sink.

- **backtrace** (`bool`, optional) – Whether or not the formatted exception should use stack trace to display local variables values. This probably should be set to `False` in production to avoid leaking sensitive data.

- **enqueue** (`bool`, optional) – Whether or not the messages to be logged should first pass through a multiprocess-safe queue before reaching the sink. This is useful while logging to a file through multiple processes.

- **catch** (`bool`, optional) – Whether or not errors occuring while sink handles logs messages should be caught or not. If `True`, an exception message is displayed on `sys.stderr` but the exception is not propagated to the caller, preventing sink from stopping working.

- ****kwargs** – Additional parameters that will be passed to the sink while creating it or while logging messages (the exact behavior depends on the sink type).

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

Parameters

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

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

- **compression** (`str` or `function`, optional) – A compression or archive format to which log files should be converted at closure.

- **delay** (`bool`, optional) – Whether or not the file should be created as soon as the sink is configured, or delayed until first logged message. It defaults to `False`.

- **mode** (`str`, optional) – The openning mode as for built-in `open()` function. It defaults to `"a"` (open the file in appending mode).

- **buffering** (`int`, optional) – The buffering policy as for built-in `open()` function. It defaults to `1` (line buffered file).

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

- ****kwargs** – Others parameters are passed to the built-in `open()` function.

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

Notes

Extended summary follows.

The sink parameter

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

- A `file-like object` like sys.stderr or open("somefile.log", "w"). Anything with a `.write()` method is considered as a file-like object. If it has a `.flush()` method, it will be automatically called after each logged message. If it has a `.stop()` method, it will be automatically called at sink termination.

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

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

- A built-in `logging.Handler` like logging.StreamHandler. In such a case, the *Loguru* records are automatically converted to the structure expected by the `logging` module.

- A `class` object that will be used to instantiate the sink using `**kwargs` attributes passed. Hence the class should instantiate objects which are therefore valid sinks.

The logged message

The logged message passed to all started 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 bellow).

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

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

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

The record dict

The record is just a Python dict, accessible from sinks by `message.record`, and usable for formatting as `"{key}"`. Some record's values are objects with two or more attributes, those can be formatted with `"{key.attr}"` (`"{key}"` would display one by default). Formatting directives like `"{key:  >3}"` also works and is specially useful for time (see bellow).

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	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 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 local time when the logging call was made	See `datetime.datetime`

The time formatting

The time field can be formatted using more human-friendly tokens. Those constitute a subset of the one used by the Pendulum library by @sdispater. To escape a token, just add square brackets around it.

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

Table 1 – continued from previous page

	Token	Output
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 contains a special `"{time}"` field that will be formatted with the current date at file creation.

The `rotation` check is made before logging each messages. 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 overwritting. This parameter accepts:

- an `int` which corresponds to the maximum file size in bytes before that the current logged file is closed and a new one started over.

- a `datetime.timedelta` which indicates the frequency of each new rotation.

- a `datetime.time` which specifies the hour when the daily rotation should occur.

- a `str` for human-friendly parametrization of one of the previously enumerated types. Examples: `"100 MB"`, `"0.5 GB"`, `"1 month 2 weeks"`, `"4 days"`, `"10h"`, `"monthly"`, `"18:00"`, `"sunday"`, `"w0"`, `"monday at 12:00"`,...

- a `function` which will be called before logging. It should accept two arguments: the logged message and the file object, and it should return `True` if the rotation should happen now, `False` otherwise.

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

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

- a `datetime.timedelta` which specifies the maximum age of files to keep.

- a `str` for human-friendly parametrization of the maximum age of files to keep. Examples: `"1 week, 3 days"`, `"2 months"`,...

- a `function` which will be called before the retention process. It should accept the list of log files as argument and process to whatever it wants (moving files, removing them, etc.).

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

- a `str` which corresponds to the compressed or archived file extension. This can be one of: `"gz"`, `"bz2"`, `"xz"`, `"lzma"`, `"tar"`, `"tar.gz"`, `"tar.bz2"`, `"tar.xz"`, `"zip"`.

- a `function` which will be called before file termination. It should accept the path of the log file as argument and process to whatever it wants (custom compression, network sending, removing it, etc.).

The color markups

To add colors to your logs, you just have to enclose your format string with the appropriate tags. This is based on the great ansimarkup library from @gvalkov. Those tags are removed if the sink don't support ansi codes.

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

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

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

Usage:

Description	Examples	
	Foreground	Background
Basic colors	`<red>`,`<r>`	`<GREEN>`, `<G>`
Light colors	`<light-blue>`,`<le>`	`<LIGHT-CYAN>`, `<LC>`
Xterm 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>`	
Shorthand (FG, BG)	`<red, yellow>`,`<r, y>`	
Shorthand (Style, FG, BG)	`<bold, cyan, white>`,`<b,,w>`,`<b,c,>`	

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

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

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

Examples

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

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

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

```
>>> from logging import StreamHandler
>>> logger.start(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, threhold=0.25)
>>> logger.start(stream_object, level="INFO")
>>> logger.start(RandomStream, level="DEBUG", seed=34567, threshold=0.5)
```

stop(*handler_id=None*)

Stop logging to a previously started sink.

> **Parameters** **handler_id** (`int` or `None`) – The id of the sink to stop, as it was returned by
> the *start()* method. If `None`, all sinks are stopped. The pre-configured sink is guaranteed
> to have the index `0`.

Examples

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

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

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

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

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

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 or not the exception should be raised again and hence propagated to the caller.

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

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

Examples

```
>>> @logger.catch
... def f(x):
...     100 / x
...
>>> def g():
...     f(10)
...     f(0)
...
>>> g()
ERROR - An error has been caught in function 'g', process 'Main' (367),␣
↪thread 'ch1' (1398):
Traceback (most recent call last, catch point marked):
  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
...
```

opt (*, *exception=None*, *record=False*, *lazy=False*, *ansi=False*, *raw=False*, *depth=0*)
Parametrize a logging call to slightly change generated log message.

Parameters

- **exception** (`bool`, `tuple` or `Exception`, optional) – It if does not evaluate as `False`, the passed exception is formatted and added to the log message. It could be an `Exception` object or a (`type`, `value`, `traceback`) tuple, otherwise the exception information is retrieved from `sys.exc_info()`.

- **record** (`bool`, optional) – If `True`, the record dict contextualizing the logging call can be used to format the message by using `{record[key]}` in the log message.

- **lazy** (`bool`, optional) – If `True`, the logging call attribute to format the message should be functions which will be called only if the level is high enough. This can be used to avoid expensive functions if not necessary.

- **ansi** (`bool`, optional) – If `True`, logged message will be colorized according to the markups it possibly contains.

- **raw** (`bool`, optional) – If `True`, the formatting of each sink will be bypassed and the message will be send as is.

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

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

Examples

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

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

```
>>> logger.opt(lazy=True).debug("If sink <= DEBUG: {x}", x=lambda: math.
↪factorial(2**5))
[18:11:19] DEBUG in '<module>' - If sink <= DEBUG:␣
↪263130836933693530167218012160000000
```

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

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

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

bind(***kwargs*)

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

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

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

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

Examples

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

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

Add, update or retrieve a logging level.

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

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

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

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

> **Parameters**
>
> - **name** (`str`) – The name of the logging level.
>
> - **no** (`int`) – The severity of the level to be added or updated.
>
> - **color** (`str`) – The color markup of the level to be added or updated.
>
> - **icon** (`str`) – The icon of the level to be added or updated.
>
> **Returns** `Level` – A namedtuple containing information about the level.

Examples

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

disable(*name*)

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

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

> **Parameters** **name** (`str`) – 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 comming 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.

> **Parameters** **name** (`str`) – 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*, *activation=None*)

Configure the core logger.

> **Parameters**
>
> - **handlers** (`list` of `dict`, optional) – A list of each handler to be started. The list should contains dicts of params passed to the *start()* function as keyword arguments. If not `None`, all previously started handlers are first stopped.
>
> - **levels** (`list` of `dict`, optional) – A list of each level to be added or updated. The list should contains 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.
>
> - **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 preprend your list with (`""`, `False`) or (`""`, `True`).
>
> **Returns** `list` of `int` – A list containing the identifiers of possibly started sinks.

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"},
...     activation=[("my_module.secret": False, "another_library.module":
→True)]
... )
[1, 2]
```

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

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

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

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

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

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

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

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

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

2.2 loguru.notifier

class Notifier
> An object to send notifications to different providers.
>
> Each method correspond to a notifications provider and return a *Notificator* parametrized according to the
> **kwargs passed. This *Notificator* should then be used through its *send()* method.
>
> You should not instantiate a *Notifier* by yourself, use from loguru import notifier instead.

Notes

The Notifier is just a tiny wrapper around the terrific notifiers library from @liiight. Refer to its documentation for more information.

Available *Notificator* are:

- *email()*

- *gitter()*

- *gmail()*

- *hipchat()*

- *join()*

- *mailgun()*

- *pagerduty()*

- *popcornnotify()*

- *pushbullet()*

- *pushover()*

- *simplepush()*

- *slack()*

- *statuspage()*

- *telegram()*

- *twilio()*

- *zulip()*

Examples

```
>>> notifier.gmail(to="dest@mail.com", host="your.server.com").send("Sending an e-
→mail.")
```

```
>>> gmail = notifier.gmail(to="dest@gmail.com", username="you@gmail.com",␣
→password="abc123")
>>> logger.start(gmail.send, level="ERROR")
```

```
>>> notificator = notifier.slack(webhook_url="http://hooks.slack.com/xxx/yyy/zzz")
>>> notificator.send("Sending Slack message...")
>>> notificator.send("...from a Python app!")
```

email(***kwargs*)

Return a *Notificator* to send messages using the Email backend.

> **Parameters**
>
> - **message** (*str*) – the content of the email message
>
> - **to** (*str*) – one or more email addresses to use
>
> - **to** – one or more email addresses to use
>
> **Other Parameters**
>
> - **subject** (*str*) – the subject of the email message (default to *"New email from 'notifiers'!"*)
>
> - **from** (*str*) – the FROM address to use in the email (default to *'docs@build-21784344-project-151228-loguru'*)
>
> - **from_** (*str*) – the FROM address to use in the email
>
> - **attachments** (*list of str*) – one or more attachments to use in the email

- **attachments** (*str*) – one or more attachments to use in the email

- **host** (*str*) – the host of the SMTP server (default to *'localhost'*)

- **port** (*int*) – the port number to use (default to *25*)

- **username** (*str*) – username if relevant

- **password** (*str*) – password if relevant

- **tls** (*bool*) – should TLS be used (default to *False*)

- **ssl** (*bool*) – should SSL be used (default to *False*)

- **html** (*bool*) – should the email be parse as an HTML file (default to *False*)

Examples

```
>>> notificator = notifier.email(
...     subject="Loguru notification",
...     to="dest@gmail.com",
...     username="user@gmail.com",
...     password="UserPassword",
...     host="smtp.gmail.com",
...     port=465,
...     ssl=True,
... )
>>> notificator.send('Notify!')
```

gitter(***kwargs*)

Return a *Notificator* to send messages using the Gitter backend.

Parameters

- **message** (*str*) – Body of the message

- **token** (*str*) – access token

- **room_id** (*str*) – ID of the room to send the notification to

Examples

```
>>> notificator = notifier.gitter(
...     token="qdp4k378twu994ss3940c35x87jbul3p6l6e32f0",
...     room_id="1935i60h67870wi4p9q0yc81",
... )
>>> notificator.send('Notify!')
```

gmail(***kwargs*)

Return a *Notificator* to send messages using the Gmail backend.

Parameters

- **message** (*str*) – the content of the email message

- **to** (*str*) – one or more email addresses to use

- **to** – one or more email addresses to use

Other Parameters

- **subject** (*str*) – the subject of the email message (default to *"New email from 'notifiers'!"*)

- **from** (*str*) – the FROM address to use in the email (default to *'docs@build-21784344-project-151228-loguru'*)

- **from_** (*str*) – the FROM address to use in the email

- **attachments** (*list of str*) – one or more attachments to use in the email

- **attachments** (*str*) – one or more attachments to use in the email

- **host** (*str*) – the host of the SMTP server (default to *'smtp.gmail.com'*)

- **port** (*int*) – the port number to use (default to *587*)

- **username** (*str*) – username if relevant

- **password** (*str*) – password if relevant

- **tls** (*bool*) – should TLS be used (default to *True*)

- **ssl** (*bool*) – should SSL be used (default to *False*)

- **html** (*bool*) – should the email be parse as an HTML file (default to *False*)

Examples

```
>>> notificator = notifier.gmail(
...     subject="Loguru notification",
...     to="dest@gmail.com",
...     username="user@gmail.com",
...     password="UserPassword",
... )
>>> notificator.send('Notify!')
```

hipchat(***kwargs*)

Return a `Notificator` to send messages using the `Hipchat` backend.

Parameters

- **room** (*str*) – The id or url encoded name of the room

- **user** (*str*) – The id, email address, or mention name (beginning with an '@') of the user to send a message to.

- **message** (*str*) – The message body

- **token** (*str*) – User token

- **id** (*str*) – An id that will help HipChat recognise the same card when it is sent multiple times

- **team_server** (*str*) – An alternate team server. Example: 'https://hipchat.corp-domain.com'

- **group** (*str*) – HipChat group name

Other Parameters

- **notify** (*bool*) – Whether this message should trigger a user notification (change the tab color, play a sound, notify mobile phones, etc). Each recipient's notification preferences are taken into account.

- **message_format** (*{'text', 'html'}*) – Determines how the message is treated by our server and rendered inside HipChat applications

- **from** (*str*) – A label to be shown in addition to the sender's name

- **color** (*{'yellow', 'green', 'red', 'purple', 'gray', 'random'}*) – Background color for message

- **attach_to** (*str*) – The message id to to attach this notification to

- **card** (*dict*) –

  - **style** (*{'file', 'image', 'application', 'link', 'media'}*) - Type of the card

  - **description** (*str*)

  - **description** (*dict*)

    * **value** (*str*)

    * **format** (*{'text', 'html'}*) - Determines how the message is treated by our server and rendered inside HipChat applications

  - **format** (*{'compact', 'medium'}*) - Application cards can be compact (1 to 2 lines) or medium (1 to 5 lines)

  - **url** (*str*) - The url where the card will open

  - **title** (*str*) - The title of the card

  - **thumbnail** (*dict*)

    * **url** (*str*) - The thumbnail url

    * **width** (*int*) - The original width of the image

    * **url@2x** (*str*) - The thumbnail url in retina

    * **height** (*int*) - The original height of the image

  - **activity** (*dict*)

    * **html** (*str*) - Html for the activity to show in one line a summary of the action that happened

    * **icon** (*str*) - The url where the icon is

    * **icon** (*dict*)

      · **url** (*str*) - The url where the icon is

      · **url@2x** (*str*) - The url for the icon in retina

  - **attributes** (*list of dict*) - List of attributes to show below the card

    * **value** (*dict*)

      · **url** (*str*) - Url to be opened when a user clicks on the label

      · **style** (*{'lozenge-success', 'lozenge-error', 'lozenge-current', 'lozenge-complete', 'lozenge-moved', 'lozenge'}*) - AUI Integrations for now supporting only lozenges

      · **label** (*str*) - The text representation of the value

      · **icon** (*str*) - The url where the icon is

      · **icon** (*dict*)

      · **url** (*str*) - The url where the icon is

      · **url@2x** (*str*) - The url for the icon in retina

    * **label** (*str*) - Attribute label

- **icon** (*str*) – The url where the icon is

- **icon** (*dict*) –

    – **url** (*str*) - The url where the icon is

    – **url@2x** (*str*) - The url for the icon in retina

Examples

```
>>> notificator = notifier.hipchat(
...     token="2YotnFZFEjr1zCsicMWpAA",
...     room=7242,
...     group="namegroup",
...     id="6492f0a6-9fa0-48cd-a3dc-2b19a0036e99",
... )
>>> notificator.send('Notify!')
```

join(***kwargs*)

Return a `Notificator` to send messages using the `Join` backend.

Parameters

- **message** (`str`) – usually used as a Tasker or EventGhost command. Can also be used with URLs and Files to add a description for those elements

- **apikey** (`str`) – user API key

Other Parameters

- **deviceId** (*str*) – The device ID or group ID of the device you want to send the message to (default to *'group.all'*)

- **deviceIds** (*list of str*) – A comma separated list of device IDs you want to send the push to

- **deviceIds** (*str*) – A comma separated list of device IDs you want to send the push to

- **deviceNames** (*list of str*) – A comma separated list of device names you want to send the push to

- **deviceNames** (*str*) – A comma separated list of device names you want to send the push to

- **url** (*str*) – A URL you want to open on the device. If a notification is created with this push, this will make clicking the notification open this URL

- **clipboard** (*str*) – some text you want to set on the receiving device's clipboard

- **file** (*str*) – a publicly accessible URL of a file

- **smsnumber** (*str*) – phone number to send an SMS to

- **smstext** (*str*) – some text to send in an SMS

- **callnumber** (*str*) – number to call to

- **interruptionFilter** (*int*) – set interruption filter mode

- **mmsfile** (*str*) – publicly accessible mms file url

- **mediaVolume** (*int*) – set device media volume

- **ringVolume** (*str*) – set device ring volume

- **alarmVolume** (*str*) – set device alarm volume

- **wallpaper** (*str*) – a publicly accessible URL of an image file

- **find** (*bool*) – set to true to make your device ring loudly

- **title** (*str*) – If used, will always create a notification on the receiving device with this as the title and text as the notification's text

- **icon** (*str*) – notification's icon URL

- **smallicon** (*str*) – Status Bar Icon URL

- **priority** (*int*) – control how your notification is displayed

- **group** (*str*) – allows you to join notifications in different groups

- **image** (*str*) – Notification image URL

Examples

```
>>> notificator = notifier.join(
...     apikey="ar0pg953181y3lc75cl8n432x6j591ro",
... )
>>> notificator.send('Notify!')
```

mailgun(***kwargs*)

Return a `Notificator` to send messages using the `Mailgun` backend.

> **Parameters**
>
> - **api_key** (`str`) – User's API key
>
> - **message** (`str`) – Body of the message. (text version)
>
> - **html** (`str`) – Body of the message. (HTML version)
>
> - **to** (`str`) – Email address of the recipient(s). Example: "Bob <bob@host.com>".
>
> - **to** – Email address of the recipient(s). Example: "Bob <bob@host.com>".
>
> - **from** (`str`) – Email address for From header
>
> - **from_** (`str`) – Email address for From header
>
> - **domain** (`str`) – MailGun's domain to use
>
> **Other Parameters**
>
> - **cc** (*list of str*) – Email address of the recipient(s). Example: "Bob <bob@host.com>".
>
> - **cc** (*str*) – Email address of the recipient(s). Example: "Bob <bob@host.com>".
>
> - **bcc** (*list of str*) – Email address of the recipient(s). Example: "Bob <bob@host.com>".
>
> - **bcc** (*str*) – Email address of the recipient(s). Example: "Bob <bob@host.com>".
>
> - **subject** (*str*) – Message subject
>
> - **attachment** (*list of str*) – File attachment
>
> - **attachment** (*str*) – File attachment

- **inline** (*list of str*) – Attachment with inline disposition. Can be used to send inline images

- **inline** (*str*) – Attachment with inline disposition. Can be used to send inline images

- **tag** (*list of str*) – Tag string

- **tag** (*str*) – Tag string

- **dkim** (*bool*) – Enables/disables DKIM signatures on per-message basis

- **deliverytime** (*str*) – Desired time of delivery. Note: Messages can be scheduled for a maximum of 3 days in the future.

- **testmode** (*bool*) – Enables sending in test mode.

- **tracking** (*bool*) – Toggles tracking on a per-message basis

- **tracking_clicks** (*{True, False, 'htmlonly'}*) – Toggles clicks tracking on a per-message basis. Has higher priority than domain-level setting. Pass yes, no or htmlonly.

- **tracking_opens** (*bool*) – Toggles opens tracking on a per-message basis. Has higher priority than domain-level setting

- **require_tls** (*bool*) – If set to True this requires the message only be sent over a TLS connection. If a TLS connection can not be established, Mailgun will not deliver the message.If set to False, Mailgun will still try and upgrade the connection, but if Mailgun can not, the message will be delivered over a plaintext SMTP connection.

- **skip_verification** (*bool*) – If set to True, the certificate and hostname will not be verified when trying to establish a TLS connection and Mailgun will accept any certificate during delivery. If set to False, Mailgun will verify the certificate and hostname. If either one can not be verified, a TLS connection will not be established.

- **headers** (*dict*) – Any other header to add

- **data** (*dict*) – attach a custom JSON data to the message

Examples

```
>>> notificator = notifier.mailgun(
...     subject="Loguru notification",
...     from_="user@gmail.com",
...     to="dest@gmail.com",
...     api_key="35a9tpnt1499o17eb14770iv2qm3775y-9258cqsa-u37b84u9",
...     domain="sandbox50v50d43fh261308q90f654p13364076.mailgun.org",
... )
>>> notificator.send('Notify!')
```

pagerduty(***kwargs*)

Return a `Notificator` to send messages using the `Pagerduty` backend.

    Parameters

- **message** (`str`) – A brief text summary of the event, used to generate the summaries/titles of any associated alerts

- **routing_key** (*str*) – The GUID of one of your Events API V2 integrations. This is the "Integration Key" listed on the Events API V2 integration's detail page

- **event_action** (*{'trigger', 'acknowledge', 'resolve'}*) – The type of event

- **source** (*str*) – The unique location of the affected system, preferably a hostname or FQDN

- **severity** (*{'critical', 'error', 'warning', 'info'}*) – The perceived severity of the status the event is describing with respect to the affected system

Other Parameters

- **dedup_key** (*str*) – Deduplication key for correlating triggers and resolves

- **timestamp** (*str*) – The time at which the emitting tool detected or generated the event in ISO 8601

- **component** (*str*) – Component of the source machine that is responsible for the event

- **group** (*str*) – Logical grouping of components of a service

- **class** (*str*) – The class/type of the event

- **custom_details** (*dict*) – Additional details about the event and affected system

- **images** (*list of dict*) –

    - **src** (*str*) - The source of the image being attached to the incident. This image must be served via HTTPS.

    - **href** (*str*) - Optional URL; makes the image a clickable link

    - **alt** (*str*) - Optional alternative text for the image

- **links** (*list of dict*) –

    - **href** (*str*) - URL of the link to be attached

    - **text** (*str*) - Plain text that describes the purpose of the link, and can be used as the link's text

popcornnotify(***kwargs*)

Return a *Notificator* to send messages using the Popcornnotify backend.

Parameters

- **message** (*str*) – The message to send

- **api_key** (*str*) – The API key

- **recipients** (*str*) – The recipient email address or phone number. Or an array of email addresses and phone numbers

- **recipients** – The recipient email address or phone number. Or an array of email addresses and phone numbers

Other Parameters subject (*str*) – The subject of the email. It will not be included in text messages.

Examples

```
>>> notificator = notifier.popcornnotify(
...     recipients="dest@gmail.com",
...     api_key="abc123456",
... )
>>> notificator.send('Notify!')
```

pushbullet(***kwargs*)

Return a *Notificator* to send messages using the Pushbullet backend.

> **Parameters**
>
> - **message** (*str*) – Body of the push
>
> - **token** (*str*) – API access token
>
> **Other Parameters**
>
> - **title** (*str*) – Title of the push
>
> - **type** (*{'note', 'link'}*) – Type of the push, one of "note" or "link" (default to *'note'*)
>
> - **type_** (*{'note', 'link'}*) – Type of the push, one of "note" or "link"
>
> - **url** (*str*) – URL field, used for type="link" pushes
>
> - **source_device_iden** (*str*) – Device iden of the sending device
>
> - **device_iden** (*str*) – Device iden of the target device, if sending to a single device
>
> - **client_iden** (*str*) – Client iden of the target client, sends a push to all users who have granted access to this client. The current user must own this client
>
> - **channel_tag** (*str*) – Channel tag of the target channel, sends a push to all people who are subscribed to this channel. The current user must own this channel.
>
> - **email** (*str*) – Email address to send the push to. If there is a pushbullet user with this address, they get a push, otherwise they get an email
>
> - **guid** (*str*) – Unique identifier set by the client, used to identify a push in case you receive it from /v2/everything before the call to /v2/pushes has completed. This should be a unique value. Pushes with guid set are mostly idempotent, meaning that sending another push with the same guid is unlikely to create another push (it will return the previously created push).

Examples

```
>>> notificator = notifier.pushbullet(
...     token="g.iwdgad0l12pu11p3mvzpada4v8fjadfh",
...     email="user@gmail.com",
... )
>>> notificator.send('Notify!')
```

pushover(***kwargs*)

Return a *Notificator* to send messages using the Pushover backend.

> **Parameters**
>
> - **user** (*str*) – the user/group key (not e-mail address) of your user (or you)
>
> - **user** – the user/group key (not e-mail address) of your user (or you)

- **message** (*str*) – your message
- **token** (*str*) – your application's API token

Other Parameters

- **title** (*str*) – your message's title, otherwise your app's name is used
- **device** (*list of str*) – your user's device name to send the message directly to that device
- **device** (*str*) – your user's device name to send the message directly to that device
- **priority** (*int*) – notification priority
- **url** (*str*) – a supplementary URL to show with your message
- **url_title** (*str*) – a title for your supplementary URL, otherwise just the URL is shown
- **sound** (*str*) – the name of one of the sounds supported by device clients to override the user's default sound choice. See *sounds* resource
- **timestamp** (*int | str*) – a Unix timestamp of your message's date and time to display to the user, rather than the time your message is received by our API
- **retry** (*int*) – how often (in seconds) the Pushover servers will send the same notification to the user. priority must be set to 2
- **expire** (*int*) – how many seconds your notification will continue to be retried for. priority must be set to 2
- **callback** (*str*) – a publicly-accessible URL that our servers will send a request to when the user has acknowledged your notification. priority must be set to 2
- **html** (*bool*) – enable HTML formatting
- **attachment** (*str*) – an image attachment to send with the message

Examples

```
>>> notificator = notifier.pushover(
...     token="ch1nisznqlttipch5e5zu3gmxo5qp7",
...     user="srpzuyopidaythfq3u1tj2fmee3ke0",
... )
>>> notificator.send('Notify!')
```

simplepush(***kwargs*)

Return a `Notificator` to send messages using the `Simplepush` backend.

Parameters

- **key** (*str*) – your user key
- **message** (*str*) – your message

Other Parameters

- **title** (*str*) – message title
- **event** (*str*) – Event ID

Examples

```
>>> notificator = notifier.simplepush(
...     key="HuxgBB",
... )
>>> notificator.send('Notify!')
```

slack(***kwargs*)

Return a `Notificator` to send messages using the `Slack` backend.

> **Parameters**
>
> - **webhook_url** (`str`) – the webhook URL to use. Register one at https://my.slack.com/services/new/incoming-webhook/
>
> - **message** (`str`) – This is the text that will be posted to the channel
>
> **Other Parameters**
>
> - **icon_url** (*str*) – override bot icon with image URL
>
> - **icon_emoji** (*str*) – override bot icon with emoji name.
>
> - **username** (*str*) – override the displayed bot name
>
> - **channel** (*str*) – override default channel or private message
>
> - **unfurl_links** (*bool*) – avoid automatic attachment creation from URLs
>
> - **attachments** (*list of dict*) –
>
>   - **title** (*str*) - Attachment title
>
>   - **author_name** (*str*) - Small text used to display the author's name
>
>   - **author_link** (*str*) - A valid URL that will hyperlink the author_name text mentioned above. Will only work if author_name is present
>
>   - **author_icon** (*str*) - A valid URL that displays a small 16x16px image to the left of the author_name text. Will only work if author_name is present
>
>   - **title_link** (*str*) - Attachment title URL
>
>   - **image_url** (*str*) - Image URL
>
>   - **thumb_url** (*str*) - Thumbnail URL
>
>   - **footer** (*str*) - Footer text
>
>   - **footer_icon** (*str*) - Footer icon URL
>
>   - **ts** (*int | str*) - Provided timestamp (epoch)
>
>   - **fallback** (*str*) - A plain-text summary of the attachment. This text will be used in clients that don't show formatted text (eg. IRC, mobile notifications) and should not contain any markup.
>
>   - **text** (*str*) - Optional text that should appear within the attachment
>
>   - **pretext** (*str*) - Optional text that should appear above the formatted data
>
>   - **color** (*str*) - Can either be one of 'good', 'warning', 'danger', or any hex color code
>
>   - **fields** (*list of dict*) - Fields are displayed in a table on the message
>
>   * **title** (*str*) - Required Field Title

* **value** (*str*) - Text value of the field. May contain standard message markup and must be escaped as normal. May be multi-line

* **short** (*bool*) - Optional flag indicating whether the *value* is short enough to be displayed side-by-side with other values

Examples

```
>>> notificator = notifier.slack(
...     webhook_url="https://hooks.slack.com/services/T5WDFU/RPB8IF/
↪UG93Wp9mgcae1V",
... )
>>> notificator.send('Notify!')
```

statuspage(***kwargs*)

Return a *Notificator* to send messages using the `Statuspage` backend.

> **Parameters**
>
> * **message** (*str*) – The name of the incident
>
> * **api_key** (*str*) – OAuth2 token
>
> * **page_id** (*str*) – Page ID
>
> **Other Parameters**
>
> * **status** (*{'investigating', 'identified', 'monitoring', 'resolved', 'scheduled', 'in_progress', 'verifying', 'completed'}*) – Status of the incident
>
> * **body** (*str*) – The initial message, created as the first incident update
>
> * **wants_twitter_update** (*bool*) – Post the new incident to twitter
>
> * **impact_override** (*{'none', 'minor', 'major', 'critical'}*) – Override calculated impact value
>
> * **component_ids** (*list of str*) – List of components whose subscribers should be notified (only applicable for pages with component subscriptions enabled)
>
> * **deliver_notifications** (*bool*) – Control whether notifications should be delivered for the initial incident update
>
> * **scheduled_for** (*str*) – Time the scheduled maintenance should begin
>
> * **scheduled_until** (*str*) – Time the scheduled maintenance should end
>
> * **scheduled_remind_prior** (*bool*) – Remind subscribers 60 minutes before scheduled start
>
> * **scheduled_auto_in_progress** (*bool*) – Automatically transition incident to 'In Progress' at start
>
> * **scheduled_auto_completed** (*bool*) – Automatically transition incident to 'Completed' at end
>
> * **backfilled** (*bool*) – Create an historical incident
>
> * **backfill_date** (*str*) – Date of incident in YYYY-MM-DD format

Examples

```
>>> notificator = notifier.statuspage(
...     api_key="fc8f938z-9250-2buh-18r2-852312zi1y42",
...     page_id="xc4tcptf84pv",
... )
>>> notificator.send('Notify!')
```

telegram(***kwargs*)

Return a *Notificator* to send messages using the `Telegram` backend.

> **Parameters**
>> * **message** (*str*) – Text of the message to be sent
>>
>> * **token** (*str*) – Bot token
>>
>> * **chat_id** (*int*) – Unique identifier for the target chat or username of the target channel (in the format @channelusername)
>>
>> * **chat_id** – Unique identifier for the target chat or username of the target channel (in the format @channelusername)
>
> **Other Parameters**
>> * **parse_mode** (*{'markdown', 'html'}*) – Send Markdown or HTML, if you want Telegram apps to show bold, italic, fixed-width text or inline URLs in your bot's message.
>>
>> * **disable_web_page_preview** (*bool*) – Disables link previews for links in this message
>>
>> * **disable_notification** (*bool*) – Sends the message silently. Users will receive a notification with no sound.
>>
>> * **reply_to_message_id** (*int*) – If the message is a reply, ID of the original message

Examples

```
>>> notificator = notifier.telegram(
...     token="110201543:AAHdqTcvCH1vGWJxfSeofSAs0K5PALDsaw",
...     chat_id=94725518,
... )
>>> notificator.send('Notify!')
```

twilio(***kwargs*)

Return a *Notificator* to send messages using the `Twilio` backend.

> **Parameters**
>> * **message** (*str*) – The text body of the message. Up to 1,600 characters long.
>>
>> * **account_sid** (*str*) – The unique id of the Account that sent this message.
>>
>> * **auth_token** (*str*) – The user's auth token
>>
>> * **to** (*str*) – The recipient of the message, in E.164 format
>>
>> * **from** (*str*) – Twilio phone number or the alphanumeric sender ID used
>>
>> * **from_** (*str*) – Twilio phone number or the alphanumeric sender ID used

- **messaging_service_id** (*str*) – The unique id of the Messaging Service used with the message

- **media_url** (*str*) – The URL of the media you wish to send out with the message

Other Parameters

- **status_callback** (*str*) – A URL where Twilio will POST each time your message status changes

- **application_sid** (*str*) – Twilio will POST MessageSid as well as MessageStatus=sent or MessageStatus=failed to the URL in the MessageStatusCallback property of this Application

- **max_price** (*float*) – The total maximum price up to the fourth decimal (0.0001) in US dollars acceptable for the message to be delivered

- **provide_feedback** (*bool*) – Set this value to true if you are sending messages that have a trackable user action and you intend to confirm delivery of the message using the Message Feedback API

- **validity_period** (*int*) – The number of seconds that the message can remain in a Twilio queue

Examples

```
>>> notificator = notifier.twilio(
...     to="+15558675310",
...     account_sid="ACw7ly6d43h6752ld32o05c1p79u7br452",
...     auth_token="n780tw69475k8w1h3z996485rccn9i25",
... )
>>> notificator.send('Notify!')
```

zulip (***kwargs*)

Return a `Notificator` to send messages using the `Zulip` backend.

Parameters

- **message** (*str*) – Message content

- **email** (*str*) – User email

- **api_key** (*str*) – User API Key

- **to** (*str*) – Target of the message

- **domain** (*str*) – Zulip cloud domain

- **server** (*str*) – Zulip server URL. Example: https://myzulip.server.com

Other Parameters

- **type** (*{'stream', 'private'}*) – Type of message to send (default to *'stream'*)

- **type_** (*{'stream', 'private'}*) – Type of message to send

- **subject** (*str*) – Title of the stream message. Required when using stream.

Examples

```
>>> notificator = notifier.zulip(
...     email="user@zulip.com",
...     to="dest@zulip.com",
...     server="https://yourZulipDomain.zulipchat.com",
...     api_key="a0b1c2d3e4f5a6b7c8d9e0f1a2b3c4d5",
... )
>>> notificator.send('Notify!')
```

class Notificator

An object to send notifications to an internally configured provider.

You should not instantiate a *Notificator* by yourself, use the `Notifier` to configure the requested notification provider instead.

Variables

- **provider** (`Provider`) – The provider object internally used to send notifications, created thanks to the `notifiers` library.

- **parameters** (`dict`) – The parameters used to configure the `Provider`.

send(*message*, ***kwargs*)

Send a notification through the internally configured provider.

Parameters

- **message** (`str`) – The message to send to the configured notifier.

- ****kwargs** – Additional parameters to override or extend configured ones before sending the message.

Returns The response from the `notifiers` provider.

2.3 loguru.parser

class Parser

An object to more easily parse generated logs.

The *Parser* provide a set of handful methods likely to be used while parsing logs for post-processing.

You should not instaniate a *Parser* by yourself, use `from loguru import parser` instead.

static cast(*_dict*, ***kwargs*)

Convert values of a dict to others defined types.

This is a convenient function used to cast dict values resulting from parsed logs from `str` to a more appropriate type.

Parameters

- **_dict** (`dict`) – The dict to which values type should be changed.

- ****kwargs** – Mapping between keys of the input _dict and the function that should be used to convert the associated value.

Returns `dict` – A copy of the input dictionnary with values converted to the appropriate type.

Example

```
>>> dico = {"some": "text", "num": "42", "date": "2018-09-12 22:23:24"}
>>> parser.cast(dico, num=int, date=lambda t: datetime.strptime(t, "%Y-%m-%d
↪%H:%M:%S")
{'some': 'text', 'num': 42, 'date': datetime.datetime(2018, 9, 12, 22, 23,
↪24)}
```

static parse(*file*, *pattern*, ***, *chunk=65536*)

Parse raw logs to 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 entries 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 alternatively 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.
>
> - **chunk** (`int`, optional) – The number of bytes read while iterating through the logs, this avoid 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()`.

Examples

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

Project Information

3.1 Contributing

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

3.1.1 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!

3.1.2 Reporting a bug

If you encountered an unexpected behavior using *Loguru*, please open a new issue so we can fix it as soon as possible! Be as specific as possible in the description of your problem so we can fix it as quickly as possible.

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
- Your development environment and local setup (IDE, Terminal, project context, anything that could be useful)
- Some minimal reproducable example

3.1.3 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 potentialy avoid unneeded work). Then, the workflow would look as follow:

1. Fork the Loguru project from Github

2. Clone the repository locally:

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

3. Activate your virtual environment:

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

4. Create a new branch from `master`:

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

5. Install *Loguru* in development mode:

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

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

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

```
$ pytest tests
```

8. Remember to update documentation if required

9. Update the `changelog.rst` file with what you improved

10. `add` and `commit` your changes, `rebase` your branch on `master`, `push` your local project:

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

11. Finally open a pull request before getting it merged!

3.2 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.

3.3 Changelog

3.3.1 0.1.0 (2018-12-07)

- Add logger
- Add notifier
- Add parser

3.3.2 0.0.1 (2017-09-04)

Initial release

Python Module Index

l
loguru, 11

Index

B

bind() (*Logger method*), 19

C

cast() (*Parser static method*), 37
catch() (*Logger method*), 17
configure() (*Logger method*), 21
critical() (*Logger method*), 22

D

debug() (*Logger method*), 22
disable() (*Logger method*), 20

E

email() (*Notifier method*), 23
enable() (*Logger method*), 21
error() (*Logger method*), 22
exception() (*Logger method*), 22

G

gitter() (*Notifier method*), 24
gmail() (*Notifier method*), 24

H

hipchat() (*Notifier method*), 25

I

info() (*Logger method*), 22

J

join() (*Notifier method*), 27

L

level() (*Logger method*), 20
log() (*Logger method*), 22
Logger (*class in loguru._logger*), 11
loguru (*module*), 11

M

mailgun() (*Notifier method*), 28

N

Notificator (*class in loguru._notifier*), 37
Notifier (*class in loguru._notifier*), 22

O

opt() (*Logger method*), 18

P

pagerduty() (*Notifier method*), 29
parse() (*Parser static method*), 38
Parser (*class in loguru._parser*), 37
popcornnotify() (*Notifier method*), 30
pushbullet() (*Notifier method*), 31
pushover() (*Notifier method*), 31

S

send() (*Notificator method*), 37
simplepush() (*Notifier method*), 32
slack() (*Notifier method*), 33
start() (*Logger method*), 11
statuspage() (*Notifier method*), 34
stop() (*Logger method*), 17
success() (*Logger method*), 22

T

telegram() (*Notifier method*), 35
trace() (*Logger method*), 22
twilio() (*Notifier method*), 35

W

warning() (*Logger method*), 22

Z

zulip() (*Notifier method*), 36