Автоматическое ведение логов приложения в Python

Автоматическое ведение логов приложения в Python

о которой является приятное ведение журналов в Python.

ски всегда лень настраивать регистратор стандартной библиотеки loging, а вместо него дение журналов имеет фундаментальное значение для каждого приложения и упрощает процесс использовать ведение журнала с самого начала и без настройки. Это так же просто, как

ого, чтобы сделать ведение логов менее болезненным и плюс ко всему добавляет ряд полезных

ительность приложения в большинстве случаев влияет незначительно, регистратор с нулевой ть его где угодно без особого беспокойства. В предстоящем выпуске критические функции ыке C.

u в виртуальное окружение:

```
# создаем виртуальное окружение
$ python3 -m venv .venv --prompt VirtualEnv
# активируем виртуальное окружение
$ source .venv/bin/activate
# ставим модуль `loguru`
(VirtualEnv) :~$ python3 -m pip install loguru --upgrade
```

Особенности:

- Готов к использованию без настройки;
- Одна функция для управления обработчиком, форматированием, фильтром;
- Упрощенное логирование в файл с ротацией/сохранением/сжатием;
- Форматирование строк с использованием фигурных скобок;
- Перехват исключений в потоках или main;
- Красивая регистрация в консоли с цветами;
- Асинхронный, потокобезопасный, многопроцессорный;
- Полностью описательные исключения;
- Структурированное ведение логов (по необходимости);
- Изменение сгенерированного сообщения журнала;
- Пользовательские уровни логирования;
- Улучшенная обработка даты и времени;
- Конфигурирование словарём, включение и выключение логирования;
- Полностью совместим со стандартным модулем logging;
- Значения по умолчанию через переменные среды;
- Легко комбинировать с модулем уведомлений.

Готов к использованию без настройки.

Основная концепция модуля loguru заключается в том, что существует один и только один регистратор. Для удобства он предварительно сконфигурирован и выводит сообщения в sys.stderr, но это поведение полностью настраивается.

```
>>> from loguru import logger
>>> logger.debug("Красивое и простое ведение журнала!")
# -19 19:48:04 | DEBUG | __main__:<module>:1 - Красивое и простое ведение журнала!
```

Объект Logger - это интерфейс, который отправляет сообщения журнала настроенным обработчикам. Все просто и интуитивно понятно.

Logger является основным объектом модуля loguru, каждая конфигурация ведения журнала и использование проходят через вызов

г только один регистратор, поэтому нет необходимости извлекать его перед использованием.

жно использовать для записи сообщений о событиях, происходящих в коде. Читая журналы
ер лучше позволяет понимать поток программы и легче отслеживать и устранять

равления обработчиком, форматированием, фильтром.

гроить форматирование логов? Как фильтровать сообщения? Как установить уровень?

отправляет сообщения журнала, добавляются с помощью метода logger.add(). Обратите
г logger можно сразу после импорта, т.к. он предварительно настроен (журналы по умолчанию
щения могут быть зарегистрированы с различными уровнями важности, и они могут быть
фигурных скобок (под капотом используется str.format()).

essage, то с ним ассоциируется "*запись*" (message.record). Эта запись представляет собой
ацию о контексте логирования: время, функция, файл, строка, поток, уровень… Ключи
format, чтобы соответствующее значение правильно отображалось в зарегистрированном
и" "{level}" вернет установленный уровень логирования). Значения некоторых ключей
кты с двумя или более атрибутами. "*Запись*" также содержит имя сценария/модуля __name__,
ые логгеры.

ь экземпляр logger, просто используйте импортированный logger. Смотрим несколько

```
from loguru import logger

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

```
from loguru import logger

def debug_only(record):
    return record["level"].name == "DEBUG"

# Другие уровни отфильтровываются
logger.add("debug.log", filter=debug_only)
```

```
from loguru import logger

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

> *Метод logger.add() следует использовать для регистрации приемников, которые конкретизируют сообщение журнала.*

Приемник сообщений регистратора (первый аргумент метода) может принимать различные формы: простая функция, строковый путь, файлоподобный объект, функция сопрограммы или встроенный обработчик.

```
# простая функция
def my_sink(message):
    record = message.record
    update_db(message, time=record["time"], level=record["level"])

logger.add(my_sink)
```

```
# строка - путь к файлу
logger.add("file_{time}.log", level="TRACE", rotation="100 MB")
```

```
# функция сопрограммы
as  publish(message):
    await api.post(message)
```

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

```
# встроенный обработчик
                                                    r
                                                    lerr), format="{message}")
```


анных обработчиков.

лить ранее добавленный обработчик, используя идентификатор, возвращенный при его
, если нужно заменить обработчик stderr по умолчанию. Для этого необходимо вызвать в коде
one), чтобы настроить объект logger под свое приложение.

e) - идентификатор удаляемого приемника, который был возвращен методом logger.add(). Если
e обработчики. Предварительно сконфигурированный обработчик (по умолчанию) гарантированно

ValueError, если handler_id не None, и нет активного обработчика с таким

```
отчики,
ию.

RNING")
```

ние в файл с ротацией/сохранением/сжатием.

истрированные сообщения в файл, то в качестве приемника нужно использовать путь к файлу
ствующий файл будет открыт/создан для записи логов). Путь также может содержать
ом будет указана текущая дата создания файла:

```
from loguru import logger

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

Если нужен регистратор сообщений в файл с его ротацией, то это также легко настроить. Модуль loguru умеет удалять старые журналы или если необходимо, то будет сжимать файлы при закрытии.

```
# Автоматическая ротация по объему файла
logger.add("file_1.log", rotation="500 MB")
# Новый файл создается каждый день в 12:00
logger.add("file_2.log", rotation="12:00")
# Автоматическая ротация 1 раз в неделю
logger.add("file_3.log", rotation="1 week")
# Удаляет лог-файлы старше 10 days
logger.add("file_X.log", retention="10 days")
# Сжимает лог-файл
logger.add("file_Y.log", compression="zip")
```

> *Аргумент rotation файловых приемников метода logger.add() принимает ограничения по размеру ИЛИ времени, но не по обоим параметрам.*
>
> *Дополнительно смотрите "Ротация по размеру и времени, разрешения лог-файлов".*
>
> *Аргумент compression может быть один из: 'gz', 'bz2', 'xz', 'lzma', 'tar', 'tar.gz', 'tar.bz2', 'tar.xz', 'zip'. Более подробно об аргументах, отвечающих за ротацию файлов логов смотрите в описании метода logger.add().*

Форматирование строк с использованием фигурных скобок.

Модуль loguru отдает предпочтение гораздо более элегантному и мощному форматированию {}, чем %. Функции ведения журнала фактически эквивалентны str.format().

```
from loguru import logger
logger.info("If you're using Python {}, prefer {feature} of course!", 3.6, feature="f-strings")
# ... | INFO     | __main__:<module>:1 - If you're using Python 3.6, prefer f-strings of course!
```

Вверх

Перехват исключений в потоках или main

Бывают случаи, когда программа неожиданно аварийно завершает работу, не регистрируя ничего в файле журнала. Также есть моменты, когда исключения, возникающие в потоках не регистрируются. Исправить такое поведение можно с помощью декоратора rует, что любая ошибка будет правильно передана в регистратор.


```
    | __main__:g:10 -  An error has been caught in function 'g' ...:
):
odule>

c80>


9d8>

    100 / x
          └ 0
ZeroDivisionError: division by zero
```

Красивая регистрация в консоли с цветами.

Модуль loguru автоматически добавляет цвета в журналы, если совместим терминал. Чтобы добавить цвета в журналы, необходимо заключить строку формата в соответствующие теги (например, какое-то сообщение). Эти теги автоматически удаляются, если приемник не поддерживает коды ansi.

Пример выделения цветом отдельных частей сообщения.

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

Специальный тег <level> (сокращенно <lvl>) преобразует регистрируемое сообщение в соответствии с настроенным стилем уровня логирования.

Теги, которые не распознаны, вызовут исключение во время синтаксического анализа. Регулярное выражение, используемое внутри для разбора тегов разметки, имеет вид: r"\\?</?((?:[fb]g\s)?[^<>\s]*)>"

Доступные теги (обратите внимание, что совместимость может различаться в зависимости от терминала):

- Цвет (сокращение): Black (k), Blue (e), Cyan (c), Green (g), Magenta (m), Red (r), White (w), Yellow (y)
- Стиль (сокращение): Bold (b), Dim (d), Normal (n), Italic (i), Underline (u), Strike (s), Reverse (v), Blink (l)

Пример настройки стилизации уровня:

```
# смотрим настройки стилизации уровня
>>> level = logger.level("ERROR")
>>> print(level)
# Level(name='ERROR', no=40, color='<red><bold>', icon='❌')

# Изменяем иконку уровня `WARNING`
>>> logger.level("WARNING", icon=r"/!\")
# Level(name='WARNING', no=30, color='<yellow><bold>', icon='/!\\')
>>> logger.warning("Updated!")
# 30 /!\ Updated!
```

Вверх

Описание	цвет текста	цвет фона

Основные цвета	<red>, <r>	<GREEN>, <G>
Светлые цвета	<light-blue>, <le>	<LIGHT-CYAN>, <LC>
8-битные цвета	<fg 86>, <fg 255>	<bg 42>, <bg 9>
	<fg #00005f>, <fg #EE1>	<bg #AF5FD7>, <bg #fff>
	<fg 0,95,0>	<bg 72,119,65>
	<bold>, <b>, <underline>, <u>	

безопасный, многопроцессорный.

истратор, по умолчанию являются потокобезопасными. Они не являются многопроцессорными, но чередь (аргумент enqueue=True) для обеспечения целостности журналов.

```
om the child")

ue=True)

ocess(target=process)

sage sent from the child
```

жно использовать, если необходимо ведение журнала в асинхронном коде.

Поддерживаются асинхронные функции, используемые в качестве приемников, и их следует ожидать с помощью logger.complete().

```
from loguru import logger
import asyncio

async def sink(message):
    # Обработка ввода-вывода...
    await asyncio.sleep(0.1)
    print(message, end="")

async def work():
    logger.info("Start")
    logger.info("End")
    await logger.complete()

>>> logger.add(sink)
# 1
>>> asyncio.run(work())
# ... | INFO     | __main__:work:2 - Start
# ... | INFO     | __main__:work:3 - End
```

Полностью описательные исключения.

Регистрация исключений, возникающих в коде, важна для отслеживания ошибок, но совершенно бесполезна, если не знать, почему произошел сбой. Модуль loguru помогает выявлять проблемы, позволяя отображать всю трассировку стека, включая значения переменных.

Выполнение следующего кода:

```
# файл `test.py`
from loguru import logger
# Осторожно, может произойти утечка конфиденциальных
# данных, например значение переменной пароля
logger.add("out.log", backtrace=True, diagnose=True)

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

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



```
    | __main__:nested:13 - What?!
):

odule>

755322f0>

sted

fc2e18>

ero
```

кция не будет работать в Python REPL по умолчанию из-за недоступности данных кадра.

Структурированное ведение журнала по мере необходимости

Для облегчения анализа журналов модуль loguru может выдавать строку JSON. Используя аргумент serialize=True, каждое сообщение журнала будет преобразовано в строку JSON перед отправкой в настроенный приемник.

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

Используя метод logger.bind(), можно конкретизировать сообщения регистратора, изменив дополнительный атрибут записи extra (словарь).

```
# внимание, в консоль выводятся сообщения логгера по умолчанию
>>> from loguru import logger
>>> 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")
```

Распечатка file.log:

```
192.168.0.1 someone Contextualize your logger easily
192.168.0.1 someone_else Inline binding of extra attribute
192.168.0.1 anybody Use kwargs to add context during formatting: anybody
```

Можно более точно контролировать журналы, комбинируя метод logger.bind() и аргумент filter:

```
# внимание, в консоль выводятся сообщения логгера по умолчанию
>>> from loguru import logger
>>> logger.add("special.log", filter=lambda record: "special" in record["extra"])
>>> logger.debug("Это сообщение не занесено в файл")
>>> logger.bind(special=True).info("Это сообщение заносится в файл!")
```

Наконец, метод logger.patch() позволяет прикреплять динамические значения к записи каждого нового сообщения:

```
>>> import sys
>>> from loguru import logger
# удаляем логгер по умолчанию
>>> r.remove(0)
>>> r.add(sys.stderr, format="{extra[utc]} {message}")
```

```
>>> from datetime import datetime, timezone
>>> logger = logger.patch(lambda record: record["extra"].update(utc=datetime.now(timezone.utc)))
# 2023-07-18 08:02:53.637937+00:00 test
```

Еще можно временно изменить контекстно-локальное состояние с помощью менеджера контекста logger.contextualize():


```
ы

t="{message} | {extra}")

")

sk_id=123):

...}
```

стратор не возвращается, дополнительный словарь extra модифицируется на месте и
зное, он использует модуль contextvars, что означает, что значения уникальны для каждого
становит свое начальное состояние после выхода из менеджера контекста.

занного сообщения журнала.

сгенерированное сообщение журнала, необходимо воспользоваться методом logger.opt().

о объединить вызовы logger.opt() в цепочку, последний имеет приоритет над остальными,
гры до их значений по умолчанию.

Пара примеров:

```
>>> from loguru import logger
# форматирования сообщения с использованием `{record[key]}`
>>> logger.opt(record=True).info("Current line is: {record[line]}")
# [18:10:33] INFO in '<module>' - Current line is: 1

# сообщение будет окрашено в соответствии с разметкой
>>> logger.opt(colors=True).warning("We got a <red>BIG</red> problem")
# [18:11:30] WARNING in '<module>' - We got a BIG problem
```

Пользовательские уровни логирования.

Модуль loguru поставляется со всеми стандартными уровнями ведения журнала, к которым добавлены logger.trace() и logger.success(). Если нужно изменить существующий, или создать какой-то свой, то это можно сделать при помощи метода logger.level().

```
# смотрим настройки уровня
>>> level = logger.level("ERROR")
>>> print(level)
# Level(name='ERROR', no=40, color='<red><bold>', icon='❌')

# создаем свой (пользовательский) уровень
>>> logger.level("CUSTOM", no=15, color="<blue>", icon="@")
# Level(name='CUSTOM', no=15, color='<blue>', icon='@')
>>> logger.log("CUSTOM", "Logging...")
# 15 @ Logging...

# Изменяем иконку уровня `WARNING`
>>> logger.level("WARNING", icon=r"/!\")
# Level(name='WARNING', no=30, color='<yellow><bold>', icon='/!\\')
>>> logger.warning("Updated!")
# 30 /!\ Updated!
```

Улучшенная обработка даты и времени.

Ст ый модуль logging раздувается такими аргументами, как datefmt или msecs, %(asctime)s и %(created)s, наивные даты
и з информации о часовом поясе, loguru исправляет не интуитивное форматирование и т. д.

Поле времени может быть отформатировано с использованием более удобных для пользователя токенов. Чтобы экранировать токен, просто заключите его в квадратные скобки, например, "[YY]" будет отображать буквально "YY".

Чтобы использовать любимое представление времени, можно установить его непосредственно в спецификаторе форматирования :HH:mm:ss} {message}". Обратите внимание, что это значение datetime представляет собой часовой пояс.


```
time:YYYY-MM-DD at HH:mm:ss} | {level} | {message}")
```

UTC, а не местное время, то можно добавить "!UTC" в самом конце формата времени, преобразует дату и время в UTC перед форматированием.

времени не используется, как, например, format="{time} {message}", то по умолчанию будет

	Token	Output
	YYYY	2000, 2001, 2002 ⋯ 2012, 2013
	YY	00, 01, 02 ⋯ 12, 13
	Q	1 2 3 4
	MMMM	January, February, March ⋯
	MMM	Jan, Feb, Mar ⋯
	MM	01, 02, 03 ⋯ 11, 12
	M	1, 2, 3 ⋯ 11, 12
	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
Se Вверх mestamp	X	1381685817, 1234567890.123

Microseconds timestamp	x	1234567890123

Конфигурирование словарём, включение и выключение логирования.

ичие между использованием loguru в библиотеке и в приложении.

ать обработчики из любого места кода. При этом рекомендуется настраивать регистратор из внутри файла, который запускает приложение.

сценариях легко, его можно настроить logger.configure() при запуске.

```
rmat": "{time} - {message}"},
rialize": True},
```

в библиотеке, то обычно не следует добавлять какой-либо обработчик методом logger.add(). г пользователь в соответствии со своими предпочтениями, и лучше в это не вмешиваться. Так ем регистраторе, то обработчики, добавленные библиотекой, также будут получать ак правило, нежелательно.

е должна создавать журналы, за исключением случаев, когда это специально запрошено. По ogger.disable() и logger.enable(). Обязательно сначала вызовите logger.disable('mylib'). я журналов библиотеки с журналами пользователя. Пользователь всегда может вызвать очет получить доступ к журналам библиотеки.

Пример:

```
# Отключение ведения логов
# Для библиотек: `my_library` - имя библиотеки `__name__`
logger.disable("my_library")
logger.info("No matter added sinks, this message is not displayed")

# Включает логгер в библиотеке
logger.enable("my_library")
logger.info("This message however is propagated to the sinks")
```

Для дополнительного удобства можно использовать модуль loguru-config для настройки регистратора непосредственно из файла конфигурации.

Полностью совместим со стандартным модулем logging.

Нужно использовать встроенный обработчик ведения журналов в качестве приемника Loguru?

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

Необходимо распространить сообщения Loguru на стандартный модуль logging?

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

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

Нужно перехватывать сообщения журналов стандартного модуля logging для приемников Loguru?

```
class InterceptHandler(logging.Handler):
    def emit(self, record):
        # Получаем соответствующий уровень `Loguru`, если он существует..
        try:
            level = logger.level(record.levelname).name
        except ValueError:
            level = record.levelno
```

```
        # Ищем вызывающего абонента, откуда поступило зарегистрированное сообщение.
        frame, depth = sys._getframe(6), 6
        while frame and frame.f_code.co_filename == logging.__file__:
```



```
exception=record.exc_info).log(level, record.getMessage())

terceptHandler()], level=0, force=True)
```

ию через переменные среды.

тратора по умолчанию? Предпочли бы другой цвет уровня DEBUG? Без проблем:

```
 <lvl>{message}</lvl>"

n>"
```

умолчанию могут быть полностью настроены. Это особенно полезно, если не нравится формат ого приемника.

d() по умолчанию может быть изменен путем установки переменной окружения LOGURU_[PARAM]. портировать $ export LOGURU_FORMAT="{time} - {message}" для настройки форматирования

же можно изменить, установив переменную окружения LOGURU_[LEVEL]_[ATTR]. Например, в "<blue>" или setx LOGURU_TRACE_ICON "🚀". Если используете команду set, не заключайте в кавычки, но при необходимости экранируйте специальный символ, например, установите LOGURU_DEBUG_COLOR=^<blue^>.

Если нужно отключить предварительно настроенный приемник, то можно установить переменной LOGURU_AUTOINIT значение False.

Легко комбинировать с модулем уведомлений.

Loguru можно легко комбинировать с отличным модулем уведомлений notifiers (должна быть установлена отдельно), чтобы получать электронные письма или уведомления в Telegram, когда программа неожиданно выходит из строя, или отправлять многие другие виды уведомлений.

```
import notifiers

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

# Отправляем одно уведомление
notifier = notifiers.get_notifier("gmail")
notifier.notify(message="The application is running!", **params)

# ИЛИ

# Можно настроить получения оповещения о каждом сообщении об ошибке
from notifiers.logging import NotificationHandler

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

Содержание раздела:

- КРАТКИЙ ОБЗОР МАТЕРИАЛА.
- Атрибут message.record модуля loguru
- Метод logger.add() модуля loguru
- ogger.configure() модуля loguru
- ogger.bind() модуля loguru

Вверх

- Метод logger.opt() модуля loguru
- Декоратор @logger.catch() модуля loguru
- Метод logger.patch() модуля loguru
- Изменение уровня обработчика модуля loguru
- u
- enable() модуля loguru
- кали, модуль loguru
- ualize() модуля loguru
- guru
- ешения лог-файлов
- ий в модуле loguru
- бработчиков, модуль loguru
- ожений модулем loguru

Вверх