

# Loggingモジュールではじめる ログ出力入門

PyCon JP 2021

2021-10-16



堤 利史 / Toshifumi Tsutsumi

GMO Pepabo, inc.

Twitter: [@tosh2230](https://twitter.com/tosh2230)

#風来のシレン #囲碁

# Contents

1. ゴールとスコープ
2. Logging モジュール探訪
3. パブリッククラウドサービスでのログ出力
4. まとめ

# Contents

1. ゴールとスコープ
2. Logging モジュール探訪
3. パブリッククラウドサービスでのログ出力
4. まとめ

# ゴール

- 利用するかどうかの判断基準を知る
- 主な構成要素を知る
- ログ出力方法と実装例を知る
  - ソースコードは GitHub にあります
  - <https://github.com/tosh223/pycon-jp-2021>

# スコープ

- Python 3.9 アプリケーション・ライブラリを対象とします
- ログを出力する話をします
- ログを受け取ったあとの話はほとんどしない予定です

出力 >> [収集, 蓄積, 加工] >> 分析

このあたり

# Contents

## 1. ゴールとスコープ

## 2. Logging モジュール探訪

2.1. Logging モジュールの概要

2.2. Logging モジュールの構成要素

2.3. Logger の性質

2.4. Logger の設定と出力 (アプリケーション編)

2.5. Logger の設定と出力 (ライブラリ編)

## 3. パブリッククラウドサービスでのログ出力

## 4. まとめ

# Contents

1. ゴールとスコープ
2. Logging モジュール探訪
  - 2.1. Logging モジュールの概要
  - 2.2. Logging モジュールの構成要素
  - 2.3. Logger の性質
  - 2.4. Logger の設定と出力 (アプリケーション編)
  - 2.5. Logger の設定と出力 (ライブラリ編)
3. パブリッククラウドサービスでのログ出力
4. まとめ



# Logging モジュールとは

- Python に標準で搭載されている
- アプリケーションやライブラリで発生した

イベントを記録するための道具

- Five Ws (who, what, when, where, why)

# イベント例

- 処理が（開始|終了）した
- 処理の待ち合わせに成功した
- 注意すべき条件分岐に入った
- 期待した状態になっていない
- エラーになった

# シンプルに出力してみる

```
>>> from logging import getLogger
```

```
>>> logger = getLogger(__name__)
```

```
>>> logger.error('This is an error!')
```

This is an error!

(print でよいのでは...?)

# print でも良いケース

- プログラムとログのライフタイムが短い
- 局所的・バッチ的なテキスト出力
- 目的がイベントの記録以外
  - デバッグ
  - ちょっとした挙動の確認

# Logging モジュールの便利ポイント

- イベント時刻・発生箇所の記録
- ログレベルの指定
- 出力フォーマットの統一
- 柔軟な出力設定
- スレッドセーフ

# 最大のメリット

標準ライブラリモジュールとしてログ記録 API が提供される利点は、**すべての Python モジュールがログ記録に参加できること**であり、これによってあなたが書くアプリケーションのログにサードパーティーのモジュールが出力するメッセージを含ませることができます。

<https://docs.python.org/ja/3/library/logging.html>

# Contents

## 1. ゴールとスコープ

## 2. Logging モジュール探訪

### 2.1. Logging モジュールの概要

### 2.2. Logging モジュールの構成要素

### 2.3. Logger の性質

### 2.4. Logger の設定と出力 (アプリケーション編)

### 2.5. Logger の設定と出力 (ライブラリ編)

## 3. パブリッククラウドサービスでのログ出力

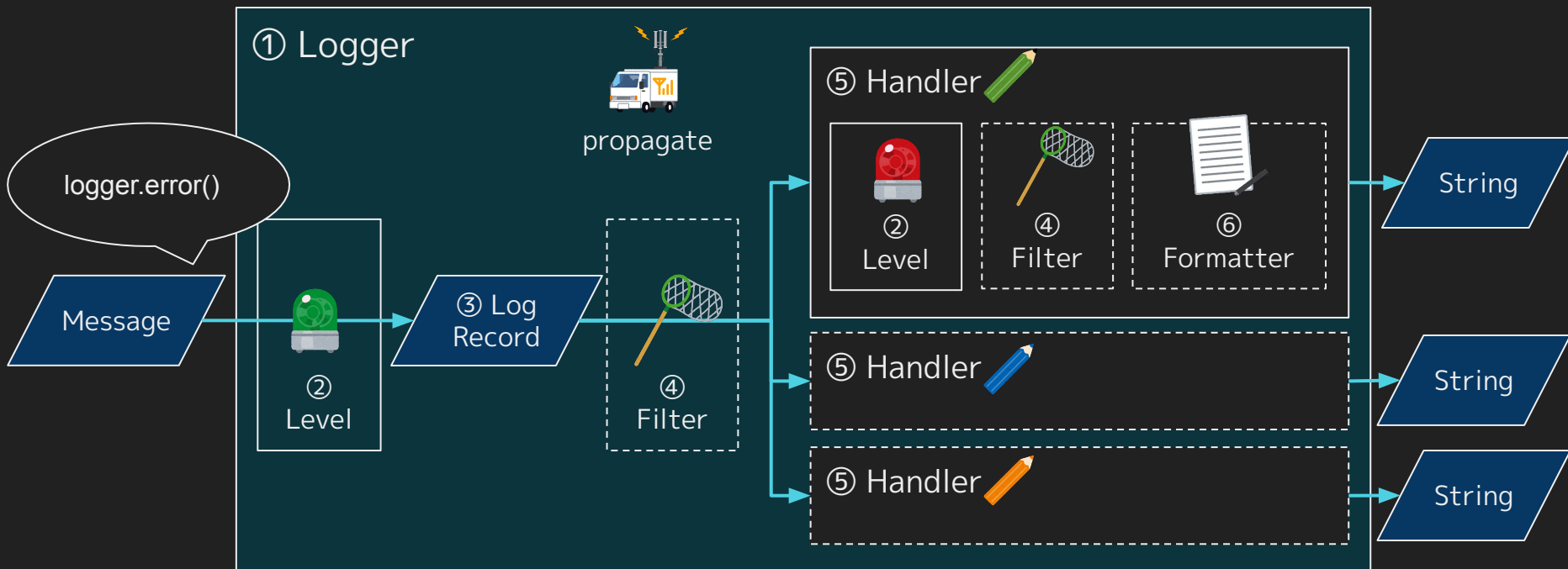
## 4. まとめ

Required

Optional

#pyconjp\_3

# 主要要素は 6つ



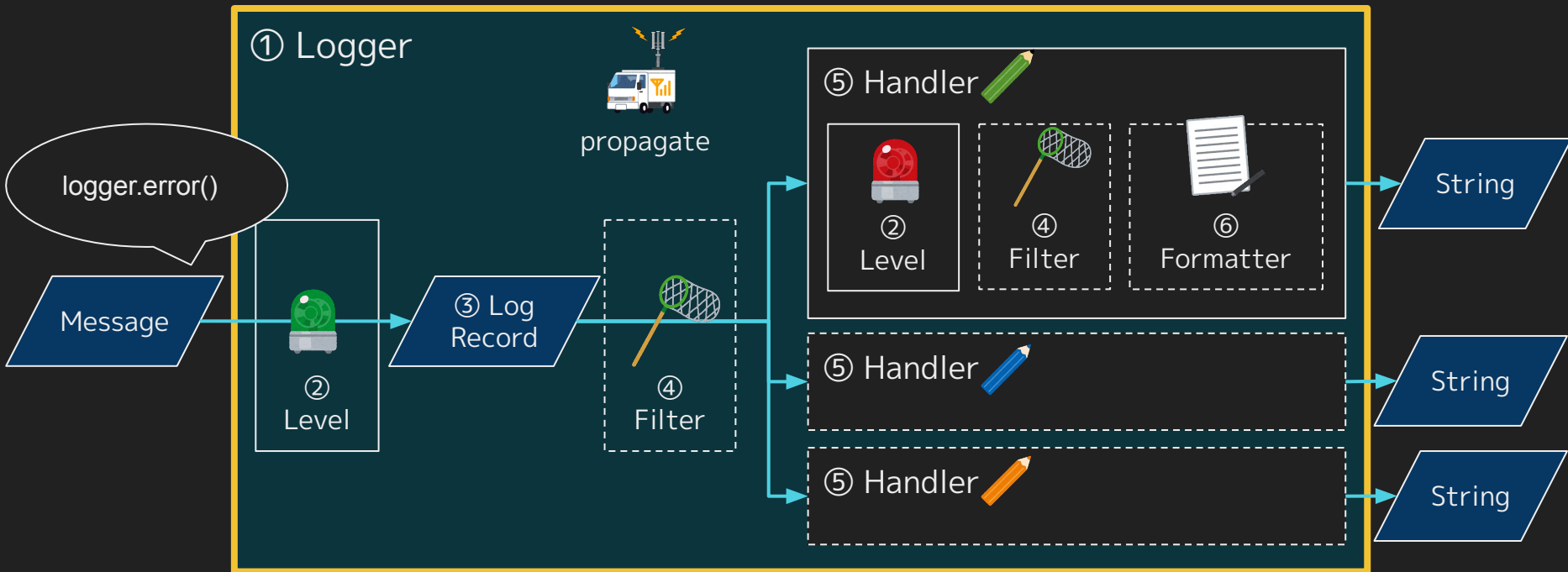


Required

Optional

#pyconjp\_3

# ① Logger



# ① Logger

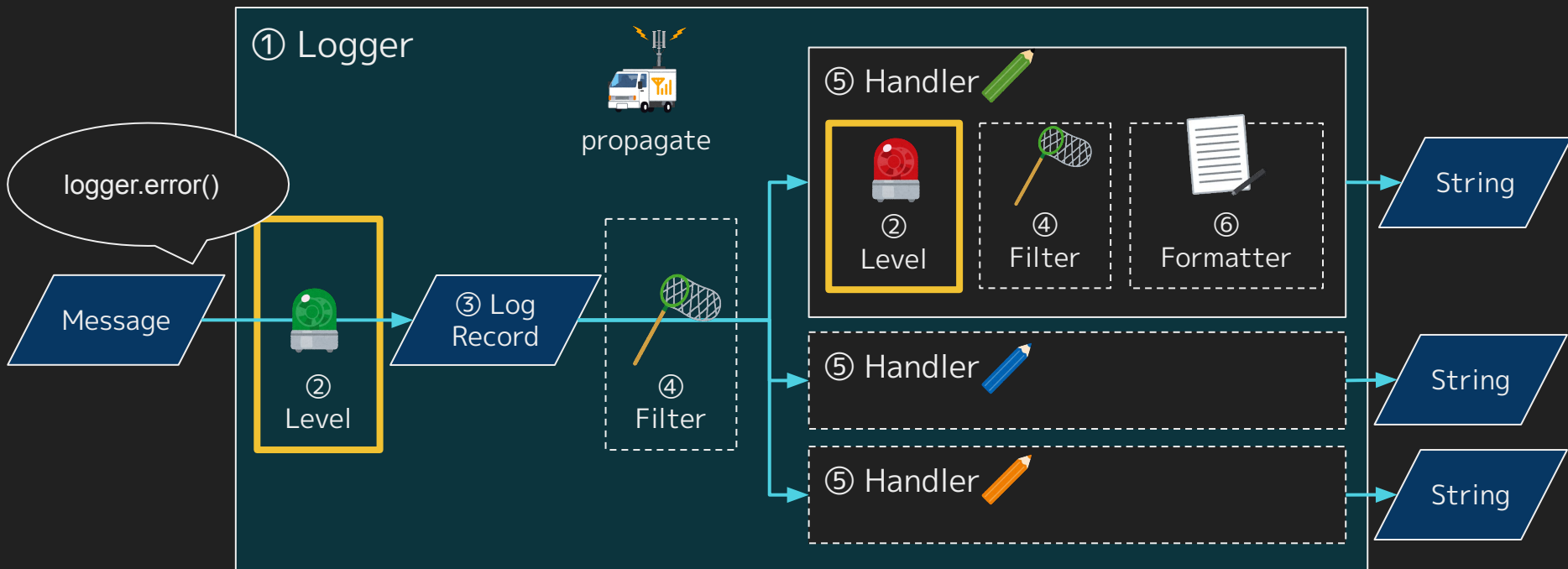
- ログを出力する主体
- メッセージを LogRecord に変換する
- Handler を使って、LogRecord を指定した書式の文字列に変換して出力する
- propagate という仕組みがある (後述)

Required

Optional

#pyconjp\_3

## ② Level




## ② Level

- Logger や Handler に設定する
- ログメッセージ出力時にイベントの重要度に応じて指定する
- 以下の条件が True のときにログが出力される

(Message level)  $\geq$  (Logger level)

and (Message level)  $\geq$  (Handler level)



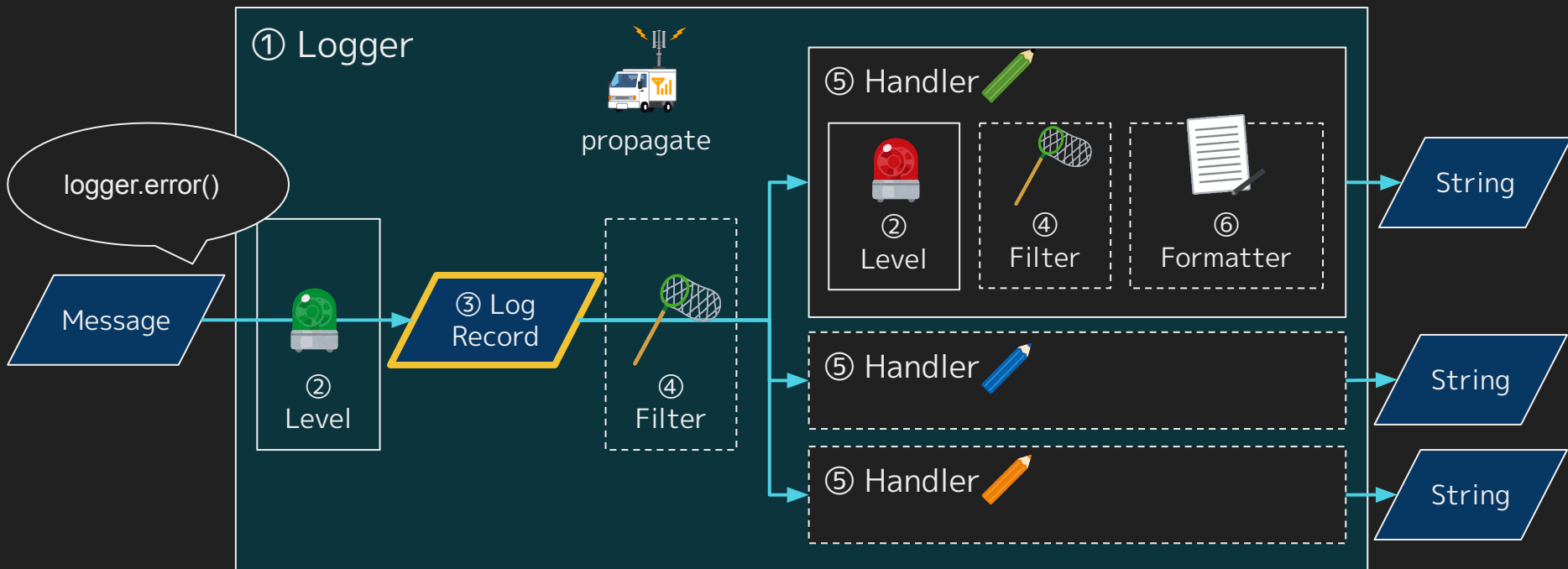
番号	名称	イベントの目安
0	NOTSET	上位ロガーに依存する。NOTSET 以外のレベルをもつ上位ロガーを見つけるか、ルートロガーに到達するまで辿っていく。
10	DEBUG	おもに問題を診断するときのみに関心があるような、詳細な情報。
20	INFO	想定された通りのことが起こったことの確認。
30	WARNING	想定外のことが起こった、または問題が近く起こりそうである (例えば、'disk space low') ことの表示。
40	ERROR	より重大な問題により、ソフトウェアがある機能を実行できないこと。
50	CRITICAL	プログラム自体が実行を続けられないことを表す、重大なエラー。

Required

Optional

#pyconjp\_3

# ③ LogRecord



### ③ LogRecord

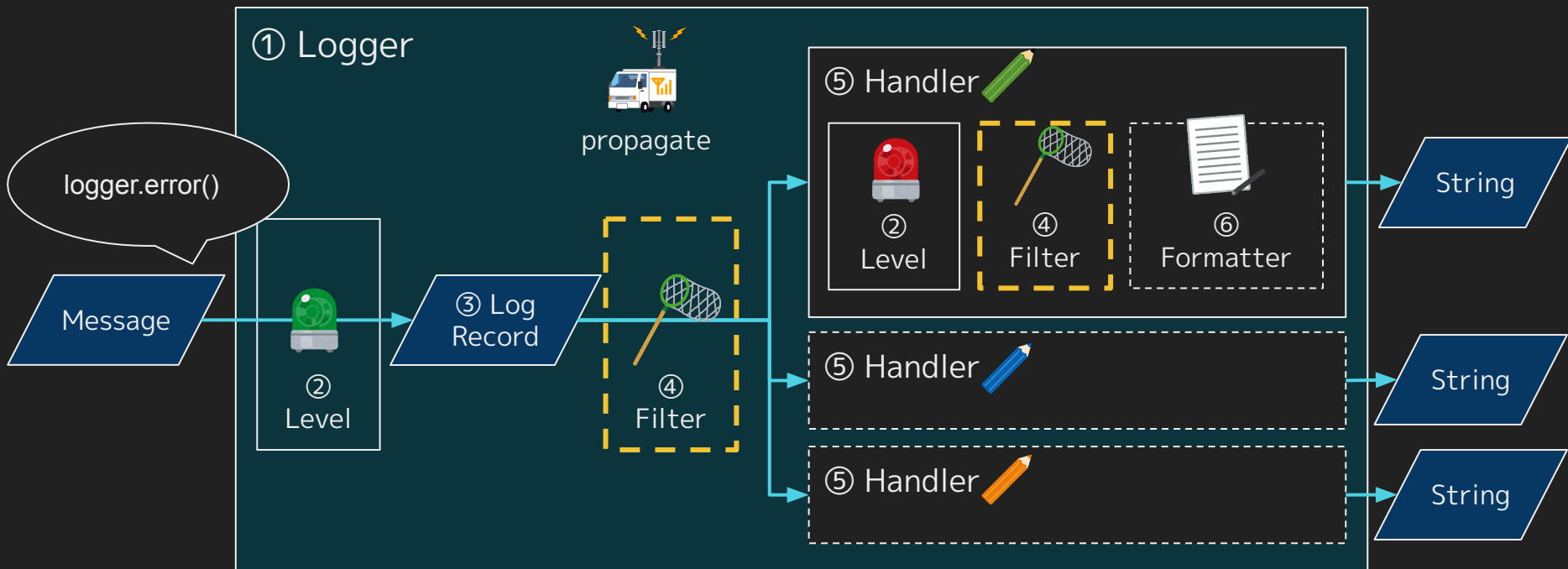
- Logger に渡されたメッセージを変換したオブジェクト
- メッセージだけでなく、イベントが発生したソースコードや実行環境に関するコンテキストを保持する
- ログに出力される時刻は、LogRecord が作成された時刻

Required

Optional

#pyconjp\_3

## ④ Filter





## ④ Filter

- 出力対象 Level と判定された LogRecord が対象
- 設定対象によってフィルタリングのタイミングが異なる
  - Logger : LogRecord が できた直後
  - Handler : Handler Level 判定の後

```
from logging import Filter
```



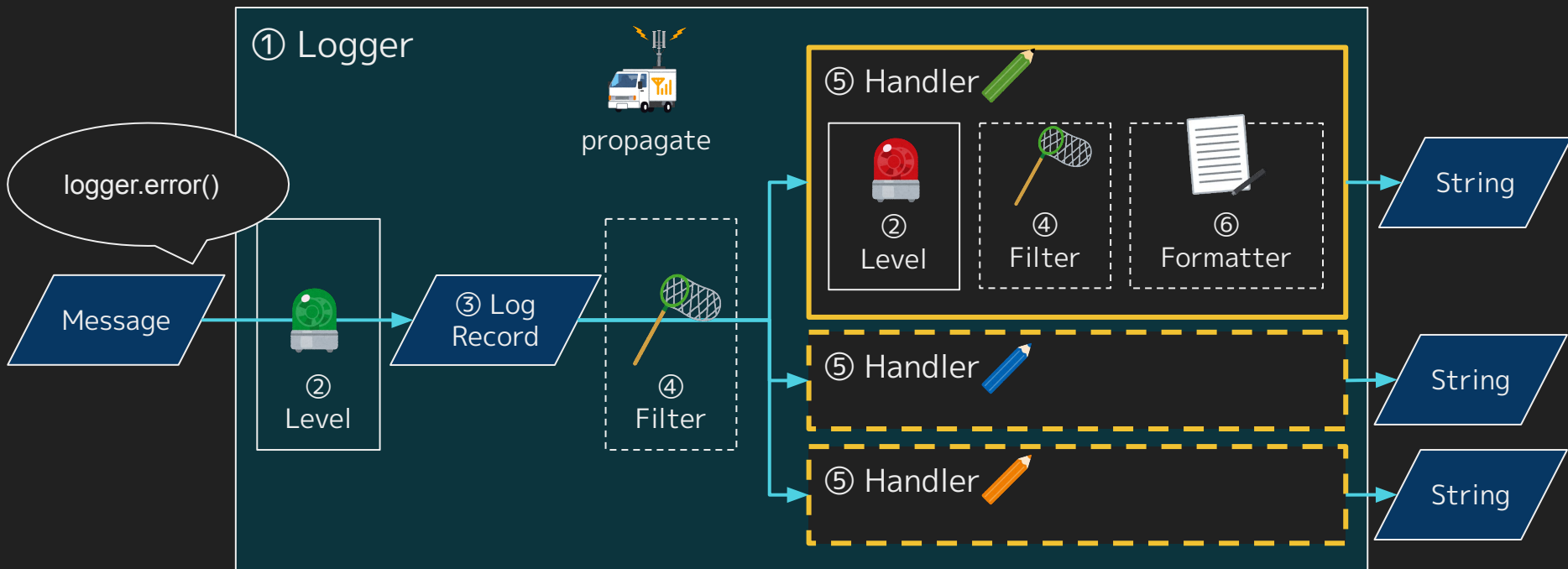
```
class CredentialsFilter(Filter):                                # Filter クラスを継承
    def __init__(self):
        pass
    def filter(self, record) -> bool:                            # filter() の返り値で出力判定
        return self.__check_message(record.getMessage())
    @staticmethod
    def __check_message(message) -> bool:
        return not message.startswith('Credentials')          # 除外するログを False にする
```

Required

Optional

#pyconjp\_3

## ⑤ Handler



## ⑤ Handler

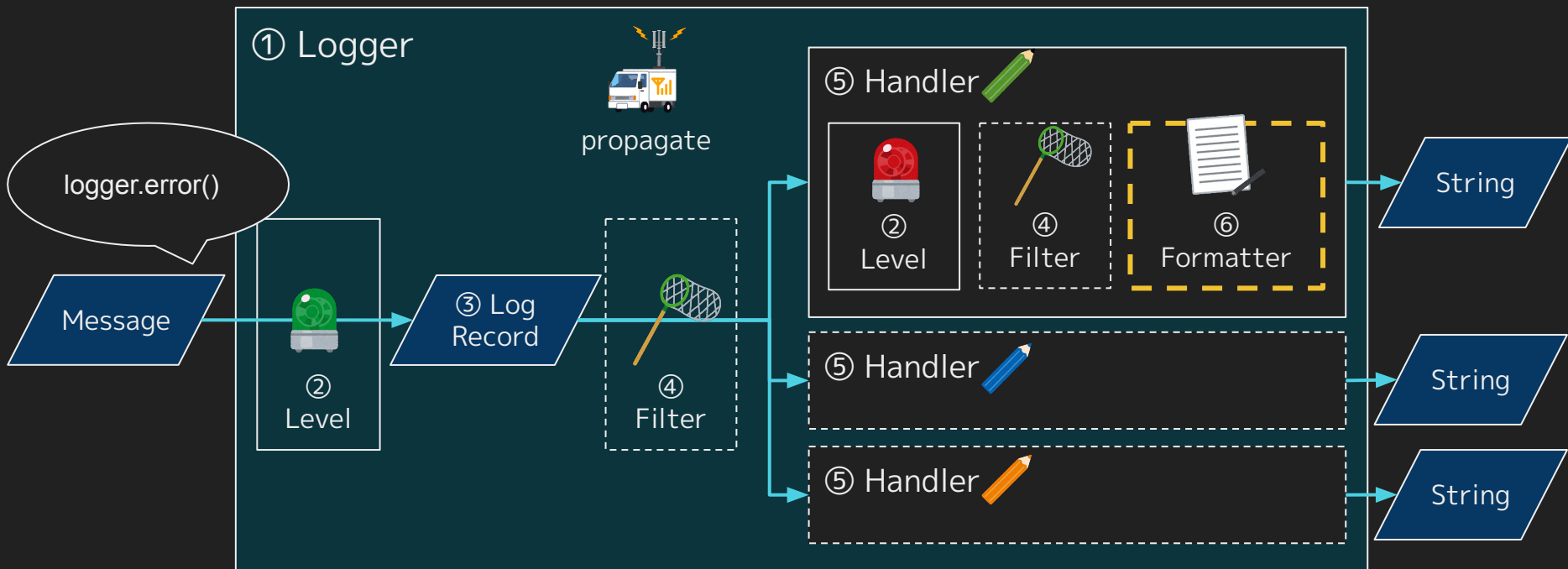
- LogRecord を特定の方法・書式で出力するための道具
- 多様な Handler が実装されている
  - StreamHandler: ストリームへの出力
  - FileHandler: ファイルへの出力
- ひとつの Logger に複数の Handler を設定できる

Required

Optional

#pyconjp\_3

## ⑥ Formatter



## ⑥ Formatter

- LogRecord を任意の文字列・日付書式に変更する
- タイムゾーンは [Formatter.converter](#) で設定
  - デフォルトは [time.localtime](#)
  - UTC に変更するには [time.gmtime](#) を設定
- メッセージフォーマットの Style は 3種類ある(次ページ)

# Message format style

- `'%'`: `%(<dict key>)s` 形式の置換文字列
  - dict key で指定できるのは [LogRecord 属性](#)
- `'{'`: [str.format\(\)](#) 互換
- `'$'`: [string.Template.substitute\(\)](#) 互換



```
>>> from logging import Formatter, StreamHandler, getLogger
>>> fmt = '%(asctime)s.%(msecs)03d %(filename)s:%(funcName)s:%(lineno)d
[% (levelname)s]%(message)s'
>>> datefmt = '%Y-%m-%d %H:%M:%S'
>>> formatter = Formatter(fmt=fmt, datefmt=datefmt, style='%')
>>> sh = StreamHandler()
>>> sh.setFormatter(formatter)           # ハンドラにフォーマッタ設定
>>> logger = getLogger()                 # ロガーの参照取得
>>> logger.addHandler(sh)                # ロガーにハンドラ設定
>>> logger.error('This is an error!')    # ロガーで記録
2021-10-16 13:50:00.000 <stdin>:<module>:1 [ERROR]This is an error!
```



# Contents

## 1. ゴールとスコープ

## 2. Logging モジュール探訪

### 2.1. Logging モジュールの概要

### 2.2. Logging モジュールの構成要素

### 2.3. Logger の性質

### 2.4. Logger の設定と出力 (アプリケーション編)

### 2.5. Logger の設定と出力 (ライブラリ編)

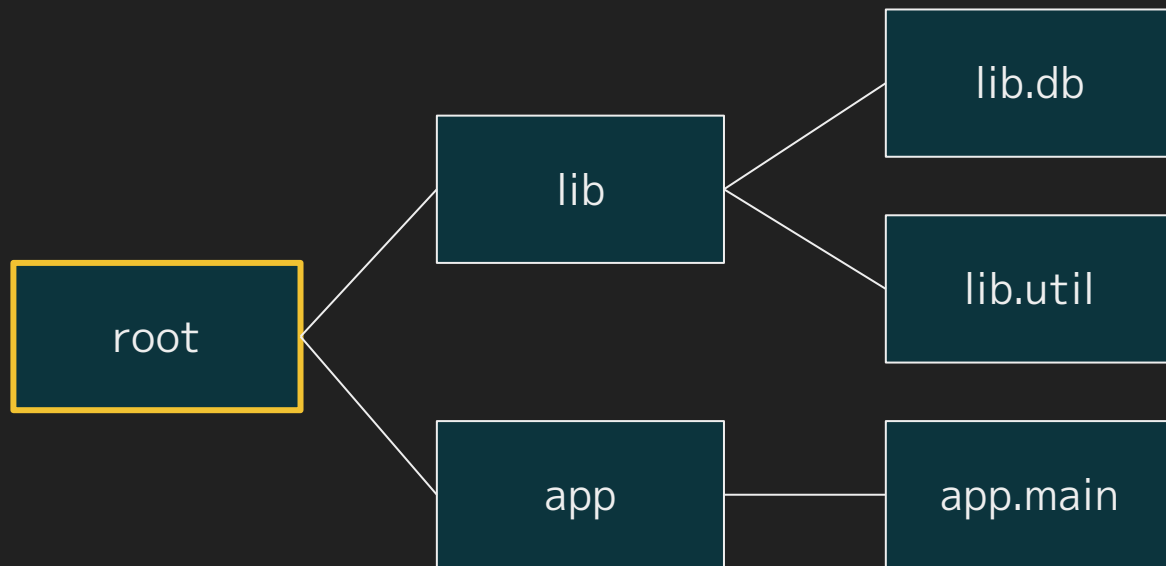
## 3. パブリッククラウドサービスでのログ出力

## 4. まとめ

# Logger の作成と命名

- Logger はモジュールごとにつくるとよい
- `logger = getLogger(__name__)` とすると、モジュールの完全修飾名と同じ名前の Logger をつくれる
- Logger がわかる = イベントがどこで起きたのかわかる
- モジュールやパッケージと同じく階層構造がある

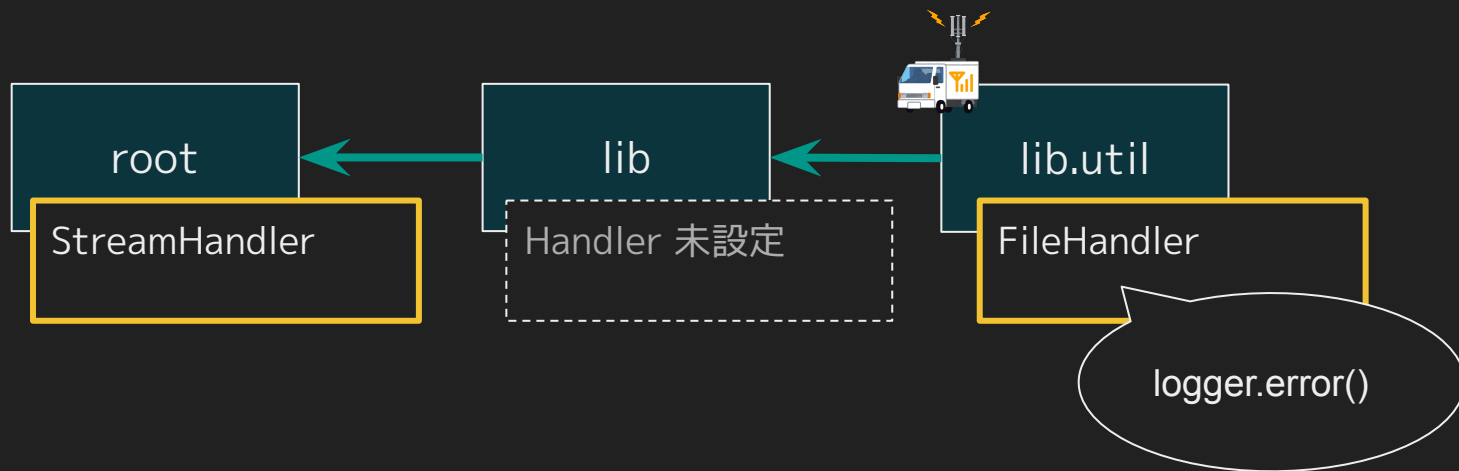
# Logger の階層構造



# root logger

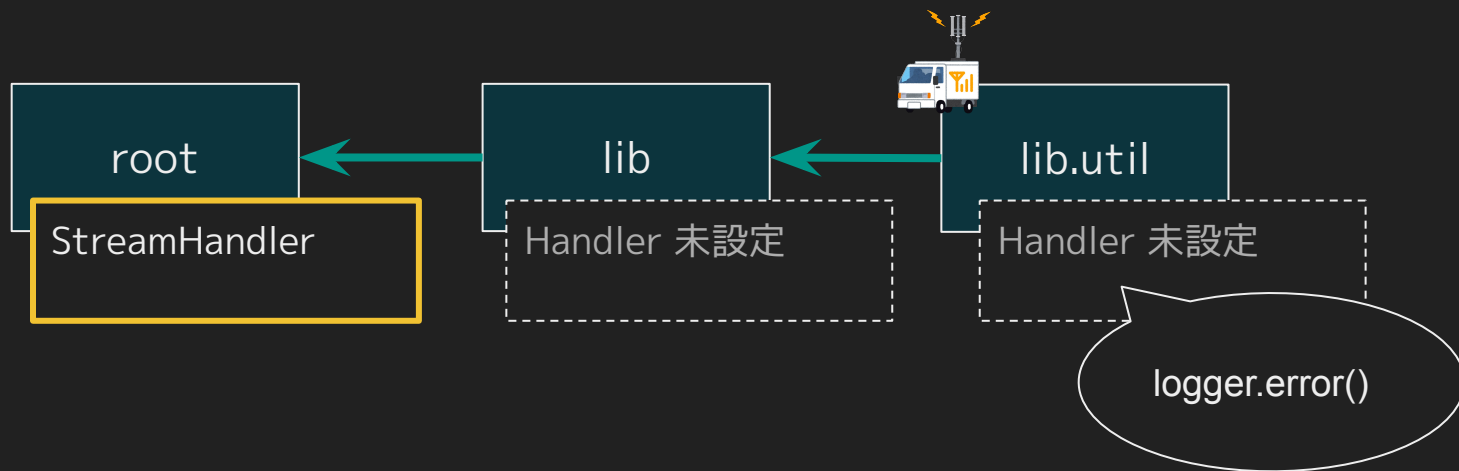
- すべての Logger の祖先
- logging モジュールが読み込まれたタイミングで自動生成
- `logging.error` など logging 直接記録は root logger を使用
- 記録時に Handler がなければ、`logging.basicConfig` (後述) が呼ばれる

# 階層構造を活かしたイベントの伝播



lib.util で メッセージを記録すると root まで伝播する  
(すべての下位 Logger が propagate=true である場合)

# 階層構造を活かしたイベントの伝播

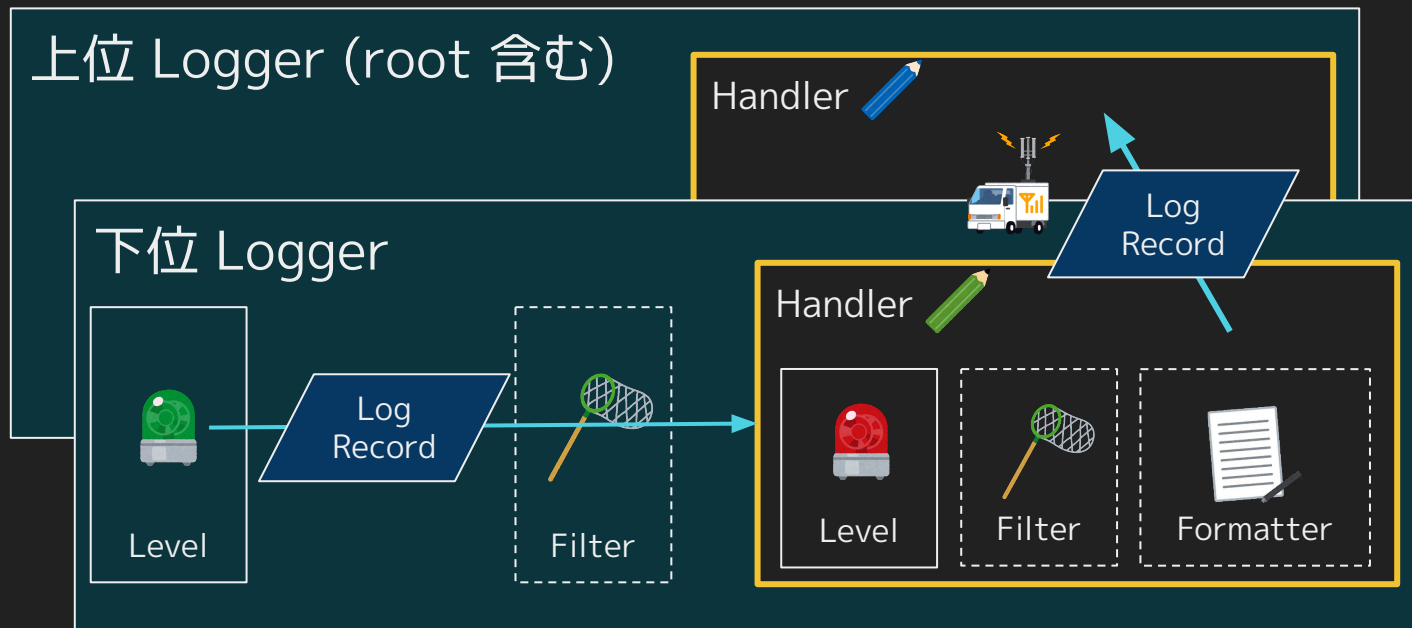


下位 Logger に Handler がなくても、同じく上位に伝播する

# 設定は root logger にまとめると便利

- エントリーポイントで、root logger に Logger 全体に適用したい設定 (=デフォルト設定) を行う
- `getLogger(__name__)` で作成した個別の Logger は “propagate=true” にする
- 特定のLogger で挙動を変えたいときだけ、個別に設定する

# 注意: Propagate 対象は Handler のみ



上位 Logger 自体に設定された Level, Filter は作用しない



# Contents

1. ゴールとスコープ
2. Logging モジュール探訪
  - 2.1. Logging モジュールの概要
  - 2.2. Logging モジュールの構成要素
  - 2.3. Logger の性質
  - 2.4. Logger の設定と出力 (アプリケーション編)
  - 2.5. Logger の設定と出力 (ライブラリ編)
3. パブリッククラウドサービスでのログ出力
4. まとめ

# 設定方法は4種類

いずれかを エントリーポイント付近で一度だけ 行う

1. コードで直接指定
2. `logging.basicConfig`
3. `logging.config.dictConfig`
4. `logging.config.fileConfig` (非推奨のため今回は省略)

# 方法1: コードで直接指定

```
sh_fmt = '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
default_fmt = Formatter(fmt=sh_fmt, style='%')           # Formatter
default_fmt.converter = gmtime                          # 時刻をUTC表示に統一
sh = StreamHandler()                                    # Handler 作成
sh.setFormatter(default_fmt)                            # Formatter を設定

logger = getLogger()                                    # ルートロガー参照を取得
logger.setLevel(DEBUG)                                  # Level を設定
logger.addHandler(sh)                                   # Handler を設定
```

## 方法2: logging.basicConfig


- root logger に対する基本設定ができる
- デフォルトでは、root logger に何らかの Handler 設定がなされている場合は反映されない(引数 force で変更可)

## 方法3: logging.config.dictConfig

- Dict から設定を読み込む
- YAML や JSON で設定ファイルをつくることできる
- デフォルトでは、呼び出す前に存在している非ルートロガーを無効化するので注意 (disable\_existing\_loggers=true)

```
import os
from logging import config
from yaml import safe_load

class DictConfig():
    def __init__(self, config_file=None):
        self.__config_file = config_file

    def set(self):
        current_dir = os.path.dirname(__file__)
        with open(current_dir + '/' + self.__config_file) as file:
            _, ext = os.path.splitext(self.__config_file)
            if ext in ['.yaml', '.yml']:
                 conf = safe_load(file)
                config.dictConfig(conf)
```

Load a yaml file

```
version: 1                                #pyconjp_3
disable_existing_loggers: False
root:
    level: WARNING
    handlers: [console]
loggers:
    __main__:
        propagate: yes
handlers:
    console:
        class: logging.StreamHandler
        level: DEBUG
        formatter: default
        stream: ext://sys.stdout
...
```

# 設定したら Logger を作って出力

```
from logging import getLogger

logger = getLogger(__name__) # ルートロガーに設定済なのでこれだけ

logger.debug('Debug')
logger.info('Info')
logger.warning('Warning')
logger.error(exception)      # logger.exception(e, exc_info=False) と同じ
logger.exception(exception)  # logger.error(e, exc_info=True) と同じ
```

# Contents

1. ゴールとスコープ
2. Logging モジュール探訪
  - 2.1. Logging モジュールの概要
  - 2.2. Logging モジュールの構成要素
  - 2.3. Logger の性質
  - 2.4. Logger の設定と出力 (アプリケーション編)
  - 2.5. Logger の設定と出力 (ライブラリ編)
3. パブリッククラウドサービスでのログ出力
4. まとめ



# ライブラリでは詳細設定をしない

- Handler/Level/Formatter/Filter の選択権は  
呼び出し側のアプリケーションにある
- ライブラリが決めるのは タイミング と メッセージ内容

# NullHandler を設定

- ライブラリのトップレベル Logger に NullHandler のみを設定するのがお作法
- NullHandler は、何もせず上位 Logger に伝播させる
- この仕組みにより 3rd party ライブラリもログ出力対象に加えることができる

<https://docs.python.org/ja/3/howto/logging.html#configuring-logging-for-a-library>

# ライブラリでの設定・出力例

```
# 設定
from logging import getLogger, NullHandler

logger = getLogger(__name__)

👉 logger.addHandler(NullHandler())
```

```
# 出力
logger.warning('Warning')
```

# Contents

1. ゴールとスコープ
2. Logging モジュール探訪
3. パブリッククラウドサービスでのログ出力
  - 3.1. 実行環境とその背景
  - 3.2. Amazon Web Services
  - 3.3. Google Cloud Platform
4. まとめ

# Contents

1. ゴールとスコープ
2. Logging モジュール探訪
3. パブリッククラウドサービスでのログ出力
  - 3.1. 実行環境とその背景
  - 3.2. Amazon Web Services
  - 3.3. Google Cloud Platform
4. まとめ

# 実行環境は用意されている

- パブリッククラウドサービスでの PaaS / FaaS では、あらかじめロギング設定がなされている
- Python の Logging モジュールを応用している
- 凝ったことをするには Logging モジュールの知識が不可欠

# 前提: The Twelve-Factor App

- モダンなアプリケーションとは、を述べたもの
- ステートレス かつ シェアードナッシング を志向
- *Factor IX*: “ログをイベントストリームとして扱う”
- ログはローカルに出さず 標準出力に書き出して収集

# Contents

1. ゴールとスコープ
2. Logging モジュール探訪
3. パブリッククラウドサービスでのログ出力
  - 3.1. 実行環境とその背景
  - 3.2. Amazon Web Services
  - 3.3. Google Cloud Platform
4. まとめ



# Amazon CloudWatch Logs

- サーバーやアプリケーションのログを収集・保存できるサービス
- AWS サービスが標準出力すると、多くは CloudWatch Logs に書き込みにいく
- AWSの各サービスが**ロギング設定を隠蔽**している


# AWS Lambda (Zip/Container)

- root logger に `LambdaLoggerHandler` というカスタムハンドラが設定されている
- Formatter で `aws_request_id` を指定できる
  - Filter で LogRecord の属性を追加するという技
  - <https://github.com/aws/aws-lambda-python-runtime-interface-client/blob/main/awslambdaric/bootstrap.py>

# Lambda の設定変更は標準と同じ

- Logging モジュールに則った設定になっているので、これまでお話しした方法で変更できる
- root logger に伝播させると(propagate=true)、前述の *aws\_request\_id* 出力を維持できるのでちょっと便利

```
def set_logger():
    level = logging.getLevelName(os.environ.get('LOG_LEVEL'))
    FMT = '%(asctime)s.%(msecs)03d\t%(aws_request_id)s\t%(filename)s:%(funcName)s:
%(lineno)d\t[%(levelname)s]%(message)s'          # フォーマット変更
    DATE_FMT = '%Y-%m-%d %H:%M:%S'
    fmt = logging.Formatter(fmt=FMT, datefmt=DATE_FMT, style='%')
    fmt.converter = time.gmtime

    root_logger = logging.getLogger()              # ルートロガー取得
    root_logger.setLevel(level)
     root_logger.handlers[0].setLevel(level)          # handlers[0] でハンドラを指定
    root_logger.handlers[0].setFormatter(fmt)
```

```
def main(event, context):
```

#pyconjp\_3

```
    logger = logging.getLogger(__name__)
```

```
    logger.info(event)
```

```
    logger.info(vars(context))          # コンテキストを出力して aws_request_id を確認
```

```
2021-10-01 13:22:26.846 18a54970-32ad-4484-9650-167ed4c7b777
```

```
lambda_function.py:main:25    [INFO]{'key1': 'value1', 'key2': 'value2', 'key3': 'value3'}
```

```
2021-10-01 13:22:26.846 18a54970-32ad-4484-9650-167ed4c7b777
```

```
lambda_function.py:main:26    [INFO]{'aws_request_id':
```

```
'18a54970-32ad-4484-9650-167ed4c7b777', 'log_group_name':
```

```
 '/aws/lambda/pycon-jp-2021_test_logging', 'log_stream_name':
```

```
 '2021/10/01/[$LATEST]0f2d935e588b4cf1be2a734049dca4e5', 'function_name':
```

```
 'pycon-jp-2021_test_logging', 'memory_limit_in_mb': '128', 'function_version': '$LATEST',
```

```
 'invoked_function_arn':
```

```
 'arn:aws:lambda:ap-northeast-1:123456789012:function:pycon-jp-2021_test_logging',
```

```
 'client_context': None, 'identity': <__main__.CognitoIdentity object at 0x7f4836e986d0>,
```

```
 '_epoch_deadline_time_in_ms': 1633094556845}
```



# Lambda powertools python

- Logging モジュールでは物足りない人に
- context を含めた形で JSON で出力できる
- *xray\_trace\_id* を簡単に出力できる
- CloudWatch Logs Insights と相性がよい

# AWS Glue: Python Script

- Spark Job ではないほう
- ログ出力先が二箇所に分かれている
  - /aws-glue/python-jobs/output
  - /aws-glue/python-jobs/error
- ログが複数行まとまって出力されることがある

# Glue python script のデフォルト設定

- /aws-glue/python-jobs/output
  - print() で出力するメッセージ
- /aws-glue/python-jobs/error
  - logging モジュールで出力するメッセージ
  - スタックトレース



# こうしたくなる気持ち

- /aws-glue/python-jobs/output
  - print() で出力するメッセージ
  - logging モジュールで出力するメッセージ
- /aws-glue/python-jobs/error
  - スタックトレース

```
args = getResolvedOptions(sys.argv, ['log_level'])
level = logging.getLevelName(args['log_level'])
FMT = '%(asctime)s.%(msecs)03d %(filename)s:%(funcName)s:%(lineno)d
[%%(levelname)s]%(message)s'
DATE_FMT = '%Y-%m-%d %H:%M:%S'
fmt = logging.Formatter(fmt=FMT, datefmt=DATE_FMT, style='%')
sh = logging.StreamHandler(stream=sys.stdout)      # stdout に向けた StreamHandler
sh.setLevel(level)                                # Level を変更
sh.setFormatter(fmt)                             # Formatter も変更

root_logger = logging.getLogger()
root_logger.handlers.clear()                      # stderr に向いているハンドラを削除
root_logger.setLevel(level)
root_logger.addHandler(sh)
```

# Contents

1. ゴールとスコープ
2. Logging モジュール探訪
3. パブリッククラウドサービスでのログ出力
  - 3.1. 実行環境とその背景
  - 3.2. Amazon Web Services
  - 3.3. Google Cloud Platform
4. まとめ

# Cloud Logging

- サーバーやアプリケーションのログを収集・保存できるサービス
- Cloud Logging クライアントライブラリを使って出力
  - <https://github.com/googleapis/python-logging>
  - サービスに応じて対応する Handler が異なる
- GCPの各サービスを**拡張するツールを提供**している

# googleapis/python-logging

- Python 標準の logging モジュールそのまま使うと Cloud Logging 独自のログレベルである 'Severity' を設定できない
- それをカバーするためのライブラリ
- 使い方は2通り、直接使用 か Logging モジュールに接続
- デフォルトのレベルは INFO

# デフォルト設定で出力してみる



```
from google.cloud.logging import Client
```

```
client = Client()
```

```
client.setup_logging()
```

# 実行サービスに応じたハンドラを  
ルートロガーにセットしてくれる

```
from logging import getLogger
```

```
logger = logging.getLogger(__name__)
```

```
logger.info('Information!')
```

# INFO 以上のログが出力対象

# 驚きの長さ(1/3)

```
{  
  "textPayload": "2021-10-02 14:33:29.246 main.py:handler:30 [INFO]Information!",  
  "insertId": "000001-6b9ed058-409f-4178-947b-444b580997c1",  
  "httpRequest": {  
    "requestMethod": "POST",  
    "requestUrl":  
"http://5caf1872csaljdhnrk8d3e68398bad9e25-dot-pcb9e111e60bda6d2p-tp.appspot.com/",  
    "userAgent": "Go-http-client/1.1",  
    "protocol": "HTTP/1.1"  
  },  
}
```

## 驚きの長さ(2/3)

```
"resource": {  
  "type": "cloud_function",  
  "labels": {  
    "function_name": "test_logging",  
    "project_id": "my-project-id",  
    "region": "us-east1"  
  }  
},  
"timestamp": "2021-10-02T14:33:29.246Z",  
"severity": "INFO",
```



## 驚きの長さ(3/3)

```
"labels": {  
  "execution_id": "ayhxxzby8ee1"  
},  
"logName": "projects/my-project-id/logs/cloudfunctions.googleapis.com%2Fcloud-functions",  
"trace": "projects/my-project-id/traces/5cb0d2b4e7997590659930dcfdd4c001",  
"sourceLocation": {  
  "file": "/workspace/main.py",  
  "line": "30",  
  "function": "handler"  
},  
"receiveTimestamp": "2021-10-02T14:33:39.442665753Z"  
}
```

# デフォルトでいいじゃん

- それでも Level や Formatter にこだわりたい あなたへ


```
import os
from logging import Formatter, getLevelName, getLogger
from google.cloud.logging.handlers import StructuredLogHandler, setup_logging

FMT =
'%(asctime)s.%(msecs)03d\t%(filename)s:%(funcName)s:%(lineno)d\t[%(levelname)s]%(message)s'

DATE_FMT = '%Y-%m-%d %H:%M:%S'

fmt = Formatter(fmt=FMT, datefmt=DATE_FMT, style='%')
level = getLevelName(os.environ.get('LOG_LEVEL'))
handler = StructuredLogHandler()
handler.setLevel(level)
handler.setFormatter(fmt)
setup_logging(handler, log_level=level)    # ルートロガーへのハンドラ設定(詳細は次ページ)
```



```
def setup_logging(  
    handler, *, excluded_loggers=EXCLUDED_LOGGER_DEFAULTS, log_level=logging.INFO  
):  
    all_excluded_loggers = set(excluded_loggers + EXCLUDED_LOGGER_DEFAULTS)  
    logger = logging.getLogger()  
    if detect_resource().type in _CLEAR_HANDLER_RESOURCE_TYPES:  
         logger.handlers.clear()                # ルートロガーの既存ハンドラをクリア  
    logger.setLevel(log_level)  
    logger.addHandler(handler)                # 新しいハンドラを追加  
    for logger_name in all_excluded_loggers:  # google.cloud.logging 関連のログ伝播を止める  
        logger = logging.getLogger(logger_name)  
        logger.propagate = False
```

[https://github.com/googleapis/python-logging/blob/main/google/cloud/logging\\_v2/handlers.py](https://github.com/googleapis/python-logging/blob/main/google/cloud/logging_v2/handlers.py) (当スライドではコメントを一部編集)

# Cloud Functions

- requirements.txt に

google-cloud-logging を指定

```
def handler(request):#pyconjp_3  
    request_json = request.get_json(silent=True)  
    logger = getLogger(__name__)  
    logger.info(f'Info {request_json}')  
    try:  
        raise ValueError("Error test")  
    except Exception as e:  
        traceback_str = traceback.format_exc().splitlines()  
        err_msg = json.dumps({  
            "errorType": e.__class__.__name__,  
            "errorMessage": e.__str__(),  
            "stackTrace": traceback_str  
        })  
        logger.error(err_msg)  
        return err_msg
```

# Cloud Run

- requirements.txt に  
google-cloud-logging を指定
- Flask を使う場合は  
app.logger.xxx 経由で出力

```
@app.errorhandler(InternalServerError)
def error_handler(e):
    return 'InternalServerError has occurred.', e.code

@app.route("/")
def test_logging():
    app.logger.debug('Debug')
    try:
        raise InternalServerError('Error test')
    except Exception as e:
        traceback_str = traceback.format_exc().splitlines()
        err_msg = json.dumps({
            "errorType": e.__class__.__name__,
            "errorMessage": e.__str__(),
            "stackTrace": traceback_str
        })
        app.logger.error(err_msg)
        abort(e.code)
```

#pyconjp\_3

# Contents

1. ゴールとスコープ
2. Logging モジュール探訪
3. パブリッククラウドサービスでのログ出力
4. まとめ

# 伝えなかったこと

- 利用するかどうかの判断基準を知る
  - 書き捨てでなければ使うと便利
- 構成要素がわかる
  - 主要な要素を図にまとめました
- 出力方法と実装例を知る
  - アプリケーション と ライブラリ でお作法が異なる
  - AWS と GCP で思想が異なる



# こんな風に出したい

- 目的が明確である
- 伝えている内容が正確な事実である
- 流量が適切である
- ネクストアクションを促している

# 参考資料

- python.org, [logging --- Python 用ロギング機能](#)
- python.org, [logging HOWTO](#)
- Adam Wiggins, [The Twelve-Factor App](#)
- @amedama, [ログ出力のための print と import logging はやめてほしい](#), Qiita
- @yomon8, [AWS Glue Python Shellでloggingを使ったログ出力について](#)
- DevelopersIO, [Lambda + Python3.7のログをJSON形式で出力してみる（行ごとに分割されない形で）](#)

# Thank you!

---

# (おまけ) Loggingモジュール関連の バージョン別変更点 まとめ

---

# Python 3.7

- [logging.Logger](#)
  - pickle 化と unpickle 化ができるようになった
- [logging.disable](#)
  - パラメータ *level* のデフォルト値が CRITICAL になった
- [logging.StreamHandler](#)
  - stream の Setter である *setStream* が追加された

# Python 3.8

- [logging.Formatter](#)
  - パラメータ *validate* が追加された (style と fmt のチェック)
- [logging.basicConfig](#)
  - 引数 *force* が追加された

# Python 3.9

- [logger.Formatter](#)
  - `default_msec_format` に `None` を指定できるようになった
- [logging.basicConfig](#)
  - 引数 `encoding`, `errors` が追加された
- [logging.FileHandler](#) (子クラスも対象)
  - パラメータ `errors` が追加された