PART I: PYTHON LOGGING BEST PRACTICES AND HOW TO INTEGRATE WITH KIBANA DASHBOARD THROUGH AWS KINESIS DATA FIREHOSE AND AMAZON ELASTICSEARCH SERVICE

Amazon Elasticsearch Service | Amazon Kinesis Data Firehose | Kibana | Python

beSharp | 15 May 2020

Applications running in production lose their ability to tell us directly what is going on under the hood, they become like black boxes and so we need a way to trace and monitor their behavior. **Logging** is by far the simplest and most straightforward approach to do so. We instruct the program during development to emit information while running that will be useful for future troubleshooting or simply for analysis.

When it comes to **Python**, the built-in module for logging is designed to fit into applications with minimal setup. Whether you've already mastered logging in Python or you are starting to develop with it, the aim of this article is to be as comprehensive as possible and to dive into methodologies to leverage logging potentials.

Note that this is a two-part article. In the second part, we will approach AWS services and use a combination of **AWS Kinesis Data Firehose** and **AWS ElasticSearch Service** to provide a way to show and debug logs on **Kibana Dashboard** as well as storing them to S3 to have long term copies at hand.

Python's logging module basics

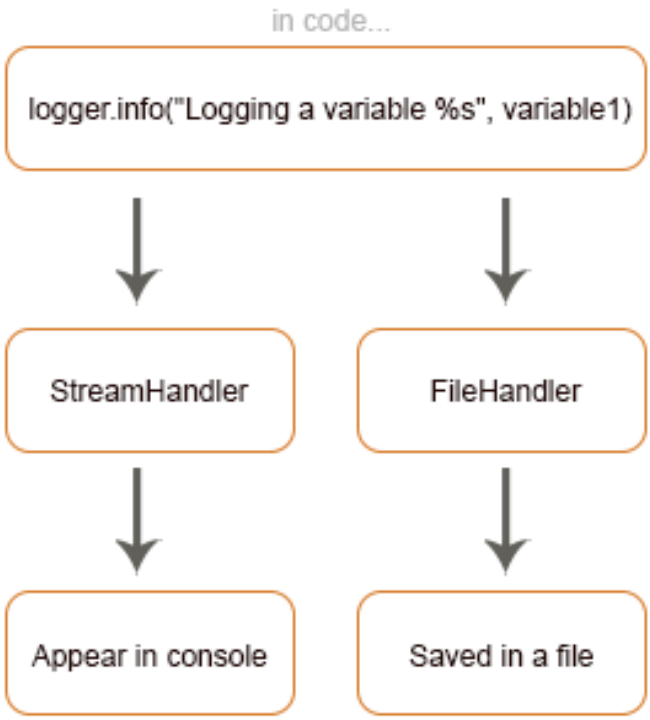
Before starting our journey, let's ask ourselves a simple question: **what is a logger?** Basically, they are objects a developer interacts with in order to print information. They are the instrument we use to tell the system what to log and how to do it.

Given an instance of a generic logger, we can deliver messages wherever we want without worrying about underneath implementation.

For example, when we write

```
logger.info("Logging a variable %s", variable1)
```

we are describing the following situation:



Python's standard library comes with a logging module that you can start using without installing anything.

In order to adhere to **best practices** which request avoiding to simply print into console whatever you want to report, Python's logging module offers multiple advantages:

- support for multi-threading;
- logging levels;
- more flexibility;
- the separation between data and formatting.

We leverage the potential of logging by separating what is logged from how it is done. When considering the "how", we should take care of three main aspects:

1. **level**: the minimum priority level of messages to log. In order of increasing severity, the available log levels are: DEBUG, INFO, WARNING, ERROR, and CRITICAL. By default, the level is set to WARNING, meaning that Python's logging module will filter out any DEBUG or INFO messages.

2. **handler**: determines where your logs will be put into. Unless specified, the logging library will use a StreamHandler sending messages to sys.stderr or, simply speaking, the console. It also handles formatting.
3. **format**: by default, the logging library will log messages in the following format: <LEVEL>:<LOGGER_NAME>:<MESSAGE>. There is, of course, the possibility to customize the look of your log lines adding custom information, we will cover that up later.

Python's logging module provides you different ways to create loggers tailored to your specific needs, combining handlers, formats, and levels.

Let's start with the simplest logger configuration method:

```
basicConfig().
```

basicConfig()

By using basicConfig() method you can **quickly configure the desired behavior** of your root logger. Other manageable approaches for larger projects include using file-based or dictionary-based logging configuration instead.

Nonetheless, basicConfig() is the preferred way to start describing how to better approach your logging.

Logging module has WARNING as its default logging level, which means that in certain situations you may lack visibility when tracing down bugs or in general when conducting a root cause analysis. By its standards the logging module, as we said before, sends logs directly to the console, but other viable options, and also recommended ones, are using a StreamHandler or a SocketHandler to stream over the network and in conjunction also using a FileHandler to log directly on disk.

Even if logging to a file has its advantages like avoiding potential network-related errors and being easy to set up, in these days and era, where applications are distributed over the network and scale fast, managing a single per-machine group of files stored locally is probably not a practical choice. Instead, this approach can be seen as a practical way to backup log files which goes in parallel with being able to centralize and monitor them over a network target.

An example of basicConfig()

Following an example that uses basicConfig() with some default parameters to log to a disk file. The options set the level to DEBUG in order to stream from debug to higher-level messages. As extra information, some simple formatting options are provided.

```
import logging

logging.basicConfig(level=logging.DEBUG,
                    filename='basic_config_test1.log',
                    format='%(asctime)s %(levelname)s:%(message)s')
```

```
def basic_config_test1():
   logging.debug("debug log test")
   logging.error("error log test")
```

Running the previous code, a new basic_config_test1.log file will be generated with the following content, except for the date, which depends obviously on the execution time.

```
2020-05-08 17:19:29,220 DEBUG:debug log test
2020-05-08 17:19:29,221 ERROR:error log test
```

NOTE: basicConfig only works the first time it is called in a runtime. If you have already configured your root logger, calling basicConfig will have no effect.

DEBUG-level logs are now visible, and logs are printed with the following custom structure:

- %(asctime)s: local time date and hour of the log row
- %(levelname)s: message severity level
- %(message)s: actual message

To complete the information here's a descriptive table of basicConfig options:

Format	Description	Example
filename	Specifies that a FileHandler should be created, using the specified filename, rather than a StreamHandler	/var/logs/logs.txt
format	Use the specified format string for the handler	"'%(asctime)s %(message)s'"
datefmt	Use the specified datetime format	"%H:%M:%S"
level	Set the root logger level to the specified level	"INFO"

This is a simple and practical way to configure small scripts. Following Python best practices we recommend managing **a logger instance for each module** of your application, but it is understandable that this can be challenging and unclean by using basicConfig() capabilities alone. So we will next focus on how to improve a basic logging solution.

Best practice: a logger instance for each module

To follow standard best practices, we can set up a logger per each module solution. One good approach which scales very well, it's easy to set up and thus should be considered when dealing with **large applications that need to scale**, is to take advantage of the built-in getLogger() method.

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

This method creates a new custom logger, different from the root one. The ß argument corresponds to the fully qualified name of the module from which this method is called. This allows you to also

take into consideration the logger's name as part of each log row by dynamically setting that name to match the one of the current modules you're working with.

It is possible to recover a logger's name property with %(name)s as shown in the following example.

```
# logging_test1.py

import logging

logging.basicConfig(level=logging.DEBUG,
                    filename='basic_config_test1.log',
                    format='%(asctime)s %(name)s %(levelname)s:%(message)s')

logger = logging.getLogger(__name__)


def basic_config_test1():
   logger.debug("debug log test 1")
   logger.error("error log test 1")


# logging_test2.py

import logging
import logging_test1

logger = logging.getLogger(__name__)



def basic_config_test2():
   logger.debug("debug log test 2")
   logger.error("error log test 2")
```

We have a **better configuration** now, each module is describing itself inside the log stream and everything is more clear, but nonetheless, the logger instantiated in the logging_test2 module will use the same configuration as the logger instantiated in the logging_test1 module, i.e. the root logger's configuration. As stated before, the second invocation of basicConfig() method will have no effect. Therefore, executing both basic_config_test1() and basic_config_test2()methods will result in the creation of a single basic_config_test1.log file with the following content.

```
2020-05-09 19:37:59,607 logging_test1 DEBUG:debug log test 1

2020-05-09 19:37:59,607 logging_test1 ERROR:error log test 1

2020-05-09 19:38:05,183 logging_test2 DEBUG:debug log test 2

2020-05-09 19:38:05,183 logging_test2 ERROR:error log test 2
```

Depending on the context of your application, using a single configuration for loggers instantiated in different modules can be not enough.

In the next section, we will see how to provide logging configuration across multiple loggers through either fileConfig() or dictConfig().

Using fileConfig() and dictConfig()

Even if basicConfig() is a quick way to start organizing your logging, file, or dictionary-based configurations offer more options to fine-tune your needs, allow for more than one logger in your application and can send your log to different destinations based on specific factors.

This is also the preferred way frameworks like Django and Flask use for setting up logging in your projects. In the next sections, we'll take a closer look at **how to set up a proper file or dictionary-based configuration.**

fileConfig()

Configuration files must adhere to this structure, containing some **key elements.**

[loggers]: the names of the loggers we need to configure.
[handlers]: the handlers we want to use for specific loggers (e.g. consoleHandler, fileHandler).
[formatters]: the formats you want to apply to each logger.

Each section of the file (named in plural) should include a comma-separated list of one or more keys to link the actual configuration:

```
[loggers]

keys=root,secondary
```

The keys are used to traverse the file and read appropriate configurations for each section. The keys are defined as [_], where the section name is logger, handler, or formatter which are predefined as said before.

A sample logging configuration file (logging.ini) is shown below.

```
[loggers]
keys=root,custom

[handlers]
keys=fileHandler,streamHandler

[formatters]
keys=formatter

[logger_root]
level=DEBUG
handlers=fileHandler

[logger_custom]
level=ERROR
handlers=streamHandler
```

```
[handler_fileHandler]
class=FileHandler
level=DEBUG
formatter=formatter
args=("/path/to/log/test.log",)

[handler_streamHandler]
class=StreamHandler
level=ERROR
formatter=formatter

[formatter_formatter]
format=%(asctime)s %(name)s - %(levelname)s:%(message)s
```

Python's best practices recommend to only maintain **one handler per logger** relying on the inheritance of child logger for properties propagation. This basically means that you just have to attach a specific configuration to a parent logger, to have it inherited by all the child loggers without having to set it up for every single one of them. A good example of this is a smart use of the root logger as the parent.

Back on track, once you have prepared a configuration file, you can attach it to a logger with these simple lines of code:

```
import logging.config

logging.config.fileConfig('/path/to/logging.ini',
                          disable_existing_loggers=False)
logger = logging.getLogger(__name__)
```

In this example, we also included disable_existing_loggers set to False, which ensure that pre-existing loggers are not removed when this snippet is executed; this is usually a good tip, as many modules declare a global logger that will be instantiated before fileConfig() is called.

dictConfig()

The logging configuration with all the properties we have described can be also defined as a standard dictionary object. This dictionary should follow the structure presented for the fileConfig() with sections covering loggers, handlers, and formatters with some global parameters.
Here's an example:

```
import logging.config


config = {
    "version": 1,
    "disable_existing_loggers": False,
    "formatters": {
        "standard": {
            "format": "%(asctime)s %(name)s %(levelname)s %(message)s",
            "datefmt": "%Y-%m-%dT%H:%M:%S%z",
        }
```

```
    },
    "handlers": {
        "standard": {
            "class": "logging.StreamHandler",
            "formatter": "standard"
        }
    },
    "loggers": {
        "": {
            "handlers": ["standard"],
            "level": logging.INFO
        }
    }
}
```

To apply this configuration simply pass it as the parameter for the dict config:

```
logging.config.dictConfig(config)
```

dictConfig() will also disable all existing loggers unless disable_existing_loggers is set to False like we did in the fileConfig() example before.

This configuration can be also stored in a YAML file and configured from there. Many developers often prefer using dictConfig() over fileConfig(), as it offers more ways to customize parameters and is more readable and maintainable.

Formatting logs the JSON way

Thanks to the way it is designed, it is easy to extend the logging module. Because our goal is to automate logging and integrate it with Kibana we want a format more suitable for adding custom properties and more compatible with custom workloads. So we want to **configure a JSON formatter.**

If we want, we can log JSON by creating a custom formatter that transforms the log records into a JSON-encoded string:

```
import logging
import json


class JsonFormatter:
    def format(self, record):
        formatted_record = dict()

        for key in ['created', 'levelname', 'pathname', 'msg']:
            formatted_record[key] = getattr(record, key)

        return json.dumps(formatted_record, indent=4)


handler = logging.StreamHandler()
handler.formatter = JsonFormatter()
```

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

logger.error("Test")
```

If you don't want to create your own custom formatter, you can rely on various libraries, developed by the Python Community, that can help you convert your logs into JSON format.
One of them is python-json-logger to convert log records into JSON.
First, install it in your environment:

```
pip install python-json-logger
```

Now update the logging configuration dictionary to customize an existing formatter or create a new formatter that will format logs in JSON like in the example below. To use the JSON formatter add pythonjsonlogger.jsonlogger.JsonFormatter as the reference for "class" property. In the formatter's "format" property, you can define the attributes you'd like to have in the log's JSON record:

```
import logging.config


config = {
   "version": 1,
   "disable_existing_loggers": False,
   "formatters": {
       "standard": {
           "format": "%(asctime)s %(name)s %(levelname)s %(message)s",
           "datefmt": "%Y-%m-%dT%H:%M:%S%z",
       },
 "json": {
    "format": "%(asctime)s %(name)s %(levelname)s %(message)s",
    "datefmt": "%Y-%m-%dT%H:%M:%S%z",
    "class": "pythonjsonlogger.jsonlogger.JsonFormatter"
 }
   },
   "handlers": {
       "standard": {
           "class": "logging.StreamHandler",
           "formatter": "json"
       }
   },
   "loggers": {
       "": {
           "handlers": ["standard"],
           "level": LogLevel.INFO.value
       }
   }
}

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

Logs that get sent to the console, through the standard handler, will get written in JSON format.

Once you've included the pythonjsonlogger.jsonlogger.JsonFormatter class in your logging configuration file, the dictConfig() function should be able to create an instance of it as long as you run the code from an environment where it can import python-json-logger module.

Add contextual information to your logs

If you need to add some context to your logs, Python's logging module gives you the possibility to **inject custom attributes** to them. An elegant way to enrich your logs involves the use of the LoggerAdapter class.

```
logger = logging.LoggerAdapter(logger, {"custom_key1": "custom_value1", "custom_key2"
: "custom_value2"})
```

This effectively adds custom attributes to all the records that go through that logger.
Note that this impacts all log records in your application, including libraries or other frameworks that you might be using and for which you are emitting logs. It can be used to log things like a unique request ID on all log lines to track requests or to add extra contextual information.

Python exception handling and tracebacks

By default, errors don't include any traceback information. The log will simply show the exception as an error, without any other information. To make sure that logging.error() prints the entire traceback, add the exc_info property with True value. To show the difference, let's try logging an exception with and without exc_info.

```
import logging.config


config = {
  "version": 1,
  "disable_existing_loggers": False,
  "formatters": {
      "standard": {
          "format": "%(asctime)s %(name)s %(levelname)s %(message)s",
          "datefmt": "%Y-%m-%dT%H:%M:%S%z",
      },
      "json": {
          "format": "%(asctime)s %(name)s %(levelname)s %(message)s",
          "datefmt": "%Y-%m-%dT%H:%M:%S%z",
          "class": "pythonjsonlogger.jsonlogger.JsonFormatter"
      }
  },
  "handlers": {
      "standard": {
          "class": "logging.StreamHandler",
          "formatter": "json"
      }
  },
  "loggers": {
      "": {
```

```
            "handlers": ["standard"],
            "level": logging.INFO
        }
    }
}

logging.config.dictConfig(config)
logger = logging.getLogger(__name__)


def test():
    try:
        raise NameError("fake NameError")
    except NameError as e:
        logger.error(e)
        logger.error(e, exc_info=True)
```

If you run test() method, it will generate the following output:

```
{"asctime": "2020-05-10T16:43:12+0200", "name": "logging_test", "levelname": "ERROR",
"message": "fake NameError"}

{"asctime": "2020-05-10T16:43:12+0200", "name": "logging_test", "levelname": "ERROR",
"message": "fake NameError", "exc_info": "Traceback (most recent call last):\n  File
 \"/Users/aleric/Projects/logging-test/src/logging_test.py\", line 39, in test\n    ra
ise NameError(\"fake NameError\")\nNameError: fake NameError"}
```

As we can see the first line doesn't provide much information about the error apart from a generic message fake NameError; instead the second line, by adding the property exc_info=True, shows the full traceback, which includes information about methods involved and line numbers to follow back and see where the exception was raised.

As an alternative you can achieve the same result by using logger.exception() from an exception handler like in the example below:

```
def test():
    try:
        raise NameError("fake NameError")
    except NameError as e:
        logger.error(e)
        logger.exception(e)
```

This can retrieve the same traceback information by using exc_info=True. This has also the benefit of setting the priority level of the log to logging.ERROR.

Regardless of which method you use to capture the traceback, having the full exception information available in your logs is critical for monitoring and troubleshooting the performance of your applications.

Capturing unhandled exceptions

Even by thinking about every possible scenario, it's guaranteed that you can never catch every exception in your application nor is it a recommended behavior but still you can log uncaught exceptions so you can investigate them later on.

An unhandled exception can be outside of a try/except block, or when you don't include the correct type in your except block.

If an exception is not handled in the method it occurs it starts bubbling up until it reaches a fitting except block or until it reaches the main block.

If it is not catched it becomes an unhandled exception and the interpreter will invoke sys.excepthook(). The information given by the method usually appears in sys.stderr but the traceback information won't get logged there if you haven't explicitly set the traceback to be shown in a non default handler (e.g. a fileHandler).

You can use **Python's standard traceback library to format the traceback** and include it in the log message.

Let's revise our example function to raise an exception not managed by our try except block to see the behavior we've described above:

```
import logging.config
import traceback

config = {
  "version": 1,
  "disable_existing_loggers": False,
  "formatters": {
      "standard": {
          "format": "%(asctime)s %(name)s %(levelname)s %(message)s",
          "datefmt": "%Y-%m-%dT%H:%M:%S%z",
      },
      "json": {
          "format": "%(asctime)s %(name)s %(levelname)s %(message)s",
          "datefmt": "%Y-%m-%dT%H:%M:%S%z",
          "class": "pythonjsonlogger.jsonlogger.JsonFormatter"
      }
  },
  "handlers": {
      "standard": {
          "class": "logging.StreamHandler",
          "formatter": "json"
      }
  },
  "loggers": {
      "": {
          "handlers": ["standard"],
          "level": logging.INFO
      }
  }
}

logging.config.dictConfig(config)
```

```
logger = logging.getLogger(__name__)


def test():
    try:
        raise OSError("fake OSError")
    except NameError as e:
        logger.error(e, exc_info=True)
    except:
        logger.error("Uncaught exception: %s", traceback.format_exc())
```

Running this code will throw a OSError exception that doesn't get handled in the try-except logic, but thanks to having the traceback code, it will get logged, as we can see in the second except clause:

```
{"asctime": "2020-05-10T17:04:05+0200", "name": "logging_test", "levelname": "ERROR",
"message": "Uncaught exception: Traceback (most recent call last):\n  File \"/Users/al
eric/Projects/logging-test/src/logging_test4.py\", line 38, in test\n    raise OSError
(\"fake OSError\")\nOSError: fake OSError\n"}
```

Logging the full traceback within each handled and unhandled exception provides **critical visibility** into errors as they occur in real time, so that you can investigate when and why they occurred.

This is the end for part one of this two-part article, in the next one, we'll cover **how to aggregate logs using Kinesis Stream and ElasticSearch** and putting all together in a highly customizable **Kibana Dashboard.**

Stay tuned 🙂

beSharp

Dal 2011 beSharp guida le aziende italiane sul Cloud. Dalla piccola impresa alla grande multinazionale, dal manifatturiero al terziario avanzato, aiutiamo le realtà più all'avanguardia a realizzare progetti innovativi in campo IT.

Get in touch

beSharp.it
proud2becloud@besharp.it