- Setup
  - Write a Logging Configuration File
  - Read the Logging Config to a Logger
  - Using the Logger
  - Lets Read Some Logs
- Making the Logger Dynamic
  - Listen for New Configurations
  - Updating Active Loggers
  - Reading the Output
- Conclusion
  - Drawbacks
  - Benefits
  - Details and Sample Code

Software visibility is vital to anyone running complex or large scale applications, and much of that visibility comes from software logs. Python provides a powerful implementation of a logging framework in its standard library. To construct what it provides into a configurable logger, we need to do a few things first.

Setup

To properly setup a customized `logger` instance, we need to create a configuration file and read from it.

Write a Logging Configuration File

Since we want to keep configurations separate from code, lets put these away from our library:

```
project
  ├── main.py
  ├┬ lib
```

```
  │ ├── __init__.py
  │ ├── color_formatter.py
  │ └── logger.py
  └┬ logger_config
    └── info.conf
```

We can build the configuration like so:

```
[loggers]
keys=root

[handlers]
keys=console

[formatters]
keys=default

[formatter_default]
class: logging.ColorFormatter
format: '$YELLOW%(asctime)s $RESET- $MAGENTA%(module)s.%(funcName)s
$RESET- $GREEN%(processName)s $RESET- $COLOR%(levelname)s $RESET- %
(message)s'

[handler_console]
class: logging.StreamHandler
formatter: default
level: INFO

[logger_root]
level: INFO
handlers: console
```

The first three entries tell Python the names of the logger, handler, and formatter we want it to use.

To support colors in our logs, we define a custom `ColorFormatter` class that replaces the log format string with the proper escape codes for readability.

This file defines a console hander that we will stream the logs into, but there are other options.

Read the Logging Config to a Logger

In `lib/logger.py`, when we construct the logger[1], we call `loggingConfig(path)`, where `path` is the path to the configuration file we want to use[2]. This function reads in the configuration file and return an instance of the logger it describes:

```
import logging
import logging.config

from .color_formatter import ColorFormatter

def build_logger() -> logging.Logger:
    logging.ColorFormatter = ColorFormatter  # type: ignore
    logging.config.fileConfig('logger_config/info.conf')
    logger = logging.getLogger()  # Root logger
    return logger


LOGGER = build_logger()
```

By creating an instance of the logger in this file, we can import that reference instead of constructing a new logger instance when we need it.

Using the Logger

In `main.py`, using the `logger` is simple:

```
import time

from lib.logger import LOGGER
```

```
while True:
    time.sleep(2)
    LOGGER.error('Oops, something weird happened.')
    LOGGER.info('Hey, something happened.')
    LOGGER.debug('Hey, something happened.')
```

This even works across processes[3]:

```
import time
from multiprocessing import Process

from lib.logger import LOGGER


def write_logs() -> None:
    while True:
        LOGGER.error('Oops, something weird happened.')
        LOGGER.info('Hey, something happened.')
        LOGGER.debug('Got http code 200.')

processes = [Process(target=write_logs) for _ in range(4)]
proc_identifier = 0
for process in processes:
    proc_identifier += 1
    process.name = f'Logging Process #{proc_identifier}'
    process.start()
write_logs()
```

Since each process spawns in its own interpreter, they each setup an instance of `LOGGER` and they each log to the same place: the parent process's standard output and standard error.

Lets Read Some Logs

Let's run this program through Logria, my logging CLI tool with `logria -e 'python main.py'`:

```
2020-10-15 18:47:04,825 - main.write_logs - MainProcess - ERROR -
Something weird happened.
2020-10-15 18:47:04,825 - main.write_logs - MainProcess - INFO - A
normal thing happened.
2020-10-15 18:47:04,843 - main.write_logs - Logging Process #1 -
ERROR - Something weird happened.
2020-10-15 18:47:04,844 - main.write_logs - Logging Process #1 - INFO
- A normal thing happened.
2020-10-15 18:47:05,349 - main.write_logs - Logging Process #2 -
ERROR - Something weird happened.
2020-10-15 18:47:05,350 - main.write_logs - Logging Process #2 - INFO
- A normal thing happened.
2020-10-15 18:47:05,853 - main.write_logs - Logging Process #3 -
ERROR - Something weird happened.
2020-10-15 18:47:05,854 - main.write_logs - Logging Process #3 - INFO
- A normal thing happened.
2020-10-15 18:47:06,365 - main.write_logs - Logging Process #4 -
ERROR - Something weird happened.
2020-10-15 18:47:06,365 - main.write_logs - Logging Process #4 - INFO
- A normal thing happened.
┌──────────────────────────────────────────────────────────────────────
──────────────────────────────┐
│No filter applied
└──────────────────────────────────────────────────────────────────────
──────────────────────────────┘
```

Since we initialize the logger with `info.conf`, we see logs at the `INFO` level and above, as expected.

If our app is running and we want to change this level to expose more detailed logs, the process needs to be restarted. While we can add gracefully exit logic and handle this, it would be far simpler to update the logging configuration after constructing it.

Making the Logger Dynamic

In order to adjust the detail level of logs on the fly, we need to make the logger smarter.

Listen for New Configurations

Python provides a method called `.listen(port: int)` in the module `logging.config`. This creates an optional `ConfigStreamHandler` instance that listens on the given port for new logging configuration files and updates the currently running configuration accordingly.

All we need to do is add logic to first validate whether a port is available, then connect to it. We set limits here so that the program does not inadvertently create too many unwanted connections.

The new logic enables the logging server when deployed to an environment, but always enables debug logging when running locally.

```
import logging
import logging.config
import os
import socket

from .color_formatter import ColorFormatter

DEFAULT_FILES = {
    'dev': 'logger_config/info.conf',
    'test': 'logger_config/info.conf',
    'prod': 'logger_config/error.conf',
}


def build_logger() -> logging.Logger:
    # Set the color prop so we can access it from the config
    logging.ColorFormatter = ColorFormatter  # type: ignore

    # Read initial config file based on environment name, default to
debug
    logging.config.fileConfig(
```

```
        DEFAULT_FILES.get(
            os.environ.get('ENV', ''),
            'logger_config/debug.conf'))

    # Create and start listener on an open port
    port = 9001
    logging_config_listener = None
    while port < 9030:
        try:
            # Check if the socket is in use
            socket_validator = socket.socket(socket.AF_INET,
                                             socket.SOCK_STREAM)
            socket_validator.connect(('localhost', port))
            socket_validator.close()
            port += 1
            continue
        # If the socket does not exist, Python raises this exception
and we know we can use it
        except ConnectionRefusedError:
            logging_config_listener = logging.config.listen(port)
            logging_config_listener.start()
            print(f'Setup logging server on port {port}')
            if port == 9030:
                print('Out of logging ports! Please increase limit!')
            break
    if logging_config_listener is None:
        raise ValueError(
            'Unable to start logging server, all ports in use!')
    logger = logging.getLogger()  # Root logger
    return logger


LOGGER = build_logger()
```

This code will loop over ports until it receives a `ConnectionRefusedError` or until the port range is exhausted. If it receives a `ConnectionRefusedError`, it means there is an available port on which to construct the `ConfigStreamHandler`.

To connect to the port, we need to write a script to read the file, convert it to the stream the `ConfigStreamHandler`

Since the listening `ConfigStreamHandler` uses `fileConfig()` internally, we must use `fileConfig()` and not `dictConfig()` when constructing the logger[4].

Updating Active Loggers

Since our loggers are all in different processes, we need to send our logger configurations to all their ports. We can use the same method as above to determine what ports are in use and reverse the logic in `update_log_levels.py`:

```
import socket
import struct
import sys

with open(sys.argv[1], 'rb') as f:
    data_to_send = f.read()

HOST = 'localhost'
PORT = 9001  # Must be the same as the port in `lib/logger.py`
UPDATED_SERVERS = 0

# Loop through all the ports on the server until we get one that does
not have a logging server
# `29` comes from the limit in the while loop in the aforementioned
file, i.e. 9001..9030
for port_num in range(PORT, PORT + 29):
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    try:
        s.connect((HOST, port_num))
        print(f'Connecting to logging server on port {port_num}...')
    except ConnectionRefusedError:
        break
    print('Sending new config...')
    s.send(struct.pack('>L', len(data_to_send)))
    s.send(data_to_send)
    UPDATED_SERVERS += 1
    s.close()
if UPDATED_SERVERS > 0:
    print('Logging configuration updated!')
else:
```

```
    print('Unable to connect to any logging servers!')
```

We can add this file here:

```
project
├── main.py
├─┬ lib
│ ├── __init__.py
│ ├── color_formatter.py
│ └── logger.py
└─┬ logger_config
  ├── debug.conf
  ├── info.conf
  ├── error.conf
  └── update_log_levels.py
```

Reading the Output

After running these it, we now see the logs, as we should:

```
2020-10-15 18:47:04,825 - main.write_logs - MainProcess - ERROR -
Something weird happened.
2020-10-15 18:47:04,825 - main.write_logs - MainProcess - INFO - A
normal thing happened.
2020-10-15 18:47:04,843 - main.write_logs - Logging Process #1 -
ERROR - Something weird happened.
2020-10-15 18:47:04,844 - main.write_logs - Logging Process #1 - INFO
- A normal thing happened.
2020-10-15 18:47:05,349 - main.write_logs - Logging Process #2 -
ERROR - Something weird happened.
2020-10-15 18:47:05,350 - main.write_logs - Logging Process #2 - INFO
- A normal thing happened.
2020-10-15 18:47:05,853 - main.write_logs - Logging Process #3 -
ERROR - Something weird happened.
2020-10-15 18:47:05,854 - main.write_logs - Logging Process #3 - INFO
- A normal thing happened.
```

```
2020-10-15 18:47:06,365 - main.write_logs - Logging Process #4 -
ERROR - Something weird happened.
2020-10-15 18:47:06,365 - main.write_logs - Logging Process #4 - INFO
- A normal thing happened.
┌──────────────────────────────────────────────────────────────────────
──────────────────────────────┐
│No filter applied
└──────────────────────────────────────────────────────────────────────
──────────────────────────────┘
```

If we open another shell and run an update command, for example, `python update_log_levels.py error.conf`, we can observe the log levels change in our Logria stream:

```
2020-10-15 19:07:24,897 - main.write_logs - MainProcess - ERROR -
Something weird happened.
2020-10-15 19:07:24,915 - main.write_logs - Logging Process #1 -
ERROR - Something weird happened.
2020-10-15 19:07:25,421 - main.write_logs - Logging Process #2 -
ERROR - Something weird happened.
2020-10-15 19:07:25,932 - main.write_logs - Logging Process #3 -
ERROR - Something weird happened.
2020-10-15 19:07:26,435 - main.write_logs - Logging Process #4 -
ERROR - Something weird happened.
┌──────────────────────────────────────────────────────────────────────
──────────────────────────────┐
│No filter applied
└──────────────────────────────────────────────────────────────────────
──────────────────────────────┘
```

Conclusion

There are several ways to use the standard library logger to work with your apps, and this one allows you to adjust the log level details without restarting the app across parallel processes. It will not interrupt compute processes and allows the application run normally while dynamically providing visibility to the process.

Drawbacks

No solution is without flaw. For this one to work, we guarantee specific ports are free. The software must wait when starting up to ensure multiple processes do not try and connect over the same port. It increases cognitive complexity, as it adds another set local network traffic to monitor.

Benefits

However, we are now able to remotely update the level we record without any downtime while the server is running and even during running compute tasks.

Details and Sample Code

You can view, run, and edit the implementation in this repl, read it on GitHub here, or download it in this zip file.

View as: PDF, Markdown

---

1. We use `fileConfig()` and not the newer `dictConfig` because later we use a method that requires `fileConfig()`.
2. There are many ways to select which configuration file we use: hardcode and manually change, use an environment variable, or write internal code to call `setLevel()` on all loggers in case of an exception. More on that later.
3. There is no need to call `.join()` in this example because the processes are all infinite loops.
4. There are drawbacks to this, but it is the method we are required to use.