Custom Python for Houlihan Lokey

John Strickler

Version 1.0, April 2023

Table of Contents

Chapter 1: Logging

Objectives

- Configure application logging
- Modify log formats
- Log to alternate destinations
- Add logging to packages

Logging overview

- Main components
  - Loggers
  - Handlers
  - Filters
  - Formatters
- Several ways to configure

"Logging" means writing messages to report what your application is doing, so they can be examined later. This is especially important for troubleshooting and debugging.

The `logging` module provides tools to implement logging. It has several components that work together.

Loggers

Loggers are called by your code to send a log message. They are the only part directly used by your code, once logging is configured. Loggers pass the log message to a *handler*.

The default logger is called the 'root' logger.

Handlers

Handlers are responsible for getting the log messages to the correct destination, such as a file or `sys.stderr`. There are many handlers provided to handle files in different ways, as well as handlers for non-file destinations.

Formatters

Formatters are responsible for the layout of the log messages.

Filters

Filters are optional, but can be used control exactly which messages get output. They allow you to specify a subset of messages within a severity level.

Configuration

Configuration specifies logging details for a particular application. You can specify which loggers, handlers, and filters will be used, as well as the format of log messages, and many other details.

Simple Logging

- Minimum configuration
  - File name
  - Minimum level
- Call level-specific methods on `logging`
  - `logging.debug()`
  - `logging.info()`
  - `logging.warning()`
  - `logging.error()`
  - `logging.critical()`

For the very simplest approach, you can just import `logging` and start calling the per-level functions such as `logging.debug()`. This will use the minimum level of `logging.WARNING` and send the output to `sys.stderr`. In general, though, you will want to log to a file.

For simple file logging, configure the default logger with a file name and minimum logging level using `logging.basicConfig()`. Then call any of the per-level methods, such as `logging.debug()` or `logging.error()`, to output a log message for that level. If the message is at or above the minimal level, it will be added to the log file. This will configure the *root logger*, which is always used when you call `logging.error()` and friends.

If no file name is specified, or if `logging.basicConfig()` is not called and there is no other configuration, the message destination will be `sys.stderr`. If no minimum severity is specified, the default level will be `logging.WARNING`. Thus, with no configuration at all, messages at level `logging.WARNING` and above will be written to the console via `sys.stderr`.

By default, log files will continue to grow, and must be manually removed or truncated. If the file does not exist, it will be created. You could also use a different handler, which will manage the log files for you.

Although `logging.basicConfig()` is sufficient for simple cases, you may want to use dictionary-based configuration which can read data from a YAML or JSON file.

NOTE Loggers are thread-safe and async-safe.

Example

logging_really_simple.py

```
import logging

# these will print to STDERR
logging.warning("I've got a bad feeling about this...")
logging.error("This is BAD")

# these won't print because default level is logging.WARNING
logging.info("The shortest president was James Madison")
logging.debug("I'm here!")
```

logging_really_simple.py

```
WARNING:root:I've got a bad feeling about this...
ERROR:root:This is BAD
```

Example

logging_simple.py

```
import logging

logging.basicConfig(
    filename='../LOGS/simple.log',
    level=logging.WARNING,
)

logging.warning('This is a warning') # message will be output
logging.debug('This message is for debugging') # message will NOT be output
logging.error('This is an ERROR') # message will be output
logging.critical('This is ***CRITICAL***') # message will be output
logging.info('The capital of North Dakota is Bismark') # message will not be output
```

../LOGS/simple.log

```
WARNING:root:This is a warning
ERROR:root:This is an ERROR
CRITICAL:root:This is ***CRITICAL***
```

Severity levels

The logger module provides 5 levels of message severity, from DEBUG to CRITICAL. When you set up a logger, you specify the minimum level of messages to be logged. For instance, if you set up the logger with the minimum level set to ERROR, then only messages at ERROR and CRITICAL levels will be logged. Setting the minimum level to DEBUG allows all messages to be logged.

Table 1. Logging Levels

Level	Value
CRITICAL	50
ERROR	40
WARNING	30
INFO	20
DEBUG	10
UNSET	0

NOTE `logging.WARN` is a deprecated alias for `logging.WARNING`, and should not be used.

Table 2. Keyword arguments for `logging.basicConfig()`

Option	Default value	Description
`filename`	`None`	Name of log file[1]
`filemode`	`'a'`	mode of file to be opened (`'w'` will truncate log when app starts)
`format`	`'%(levelname)s %(name)s %(message)'`	Format for one log entry
`datefmt`	`None`	Format for the `(asctime)s` directive; uses same format as `time.strftime()`
`style`	`'%'`	Style for format string. Choices are `'%'`, `'{'`, or ``` ``.
`level`	`logger.WARNING`	Set root logger level
`stream`	`sys.stderr`	Stream log messages will be written to (any writable file-like object)[1]
`handlers`	`None`	Iterable of handler objects[1]
`force`	`False`	Remove and close any existing handlers
`encoding`	`'utf-8'`	Encoding for log file
`errors`	`'backslashreplace'`	Value controls how encoding errors are handled.

[1] One, and only one, of `filename`, `stream` or `handlers` must be provided.

Formatting log entries

- Add `format=format` to `logging.basicConfig()` parameters
- Format is a string containing directives and other text

To format log entries, provide a format parameter to the `.logging.basicConfig()` method. This format will be a string contain special directives (i.e. placeholders) and, optionally, other text. The directives are replaced with logging information; other data is left as-is.

Directives are in the form `%(item)type`, where item is the data field, and *type* is the data type.

Formatting dates

The `%(asctime)s` directive can be formatted using the date format directives from `datetime.strftime`. Specify the `datefmt` option to `logging.basicConfig()`. === Formatting messages You can insert values into log messages by using the `%s` placeholder. When you call a log method such as `logging.warning()`, the first argument is the message, and any other arguments are inserted into the placeholders.

```
index = 12
try:
    print(mydata[index])
except IndexError as err:
    logging.error("Invalid index: %s", index)
```

Example

logging_formatted.py

```
import logging

logging.basicConfig(
    format='%(levelname)s %(name)s %(asctime)s %(filename)s %(lineno)d %(message)s', #
set the format for log entries
    datefmt="%x-%X",
    filename='../LOGS/formatted.log',
    level=logging.INFO,
)

logging.info("this is information")
logging.warning("this is a warning")
logging.error("this is an ERROR")
value = 38.7
logging.error("Invalid value %s", value)
logging.info("this is information")
logging.critical("this is critical")
```

../LOGS/formatted.log

```
INFO root 04/19/23-10:01:52 logging_formatted.py 11 this is information
WARNING root 04/19/23-10:01:52 logging_formatted.py 12 this is a warning
ERROR root 04/19/23-10:01:52 logging_formatted.py 13 this is an ERROR
ERROR root 04/19/23-10:01:52 logging_formatted.py 15 Invalid value 38.7
INFO root 04/19/23-10:01:52 logging_formatted.py 16 this is information
CRITICAL root 04/19/23-10:01:52 logging_formatted.py 17 this is critical
```

Table 3. Log entry formatting directives

Directive	Description
`%(name)s`	Name of the logger (logging channel)
`%(levelno)s`	Numeric logging level for the message (DEBUG, INFO, WARNING, ERROR, CRITICAL)
`%(levelname)s`	Text logging level for the message ("DEBUG", "INFO", "WARNING", "ERROR", "CRITICAL")
`%(pathname)s`	Full pathname of the source file where the logging call was issued (if available)
`%(filename)s`	Filename portion of pathname
`%(module)s`	Module (name portion of filename)
`%(lineno)d`	Source line number where the logging call was issued (if available)
`%(funcName)s`	Function name
`%(created)f`	Time when the LogRecord was created (time.time() return value)
`%(asctime)s`	Textual time when the LogRecord was created
`%(msecs)d`	Millisecond portion of the creation time
`%(relativeCreated)d`	Time in milliseconds when the LogRecord was created, relative to the time the logging module was loaded (typically at application startup time)
`%(thread)d`	Thread ID (if available)
`%(threadName)s`	Thread name (if available)
`%(process)d`	Process ID (if available)
`%(message)s`	The result of record.getMessage(), computed just as the record is emitted

Table 4. Date Format Directives

Directive	Meaning
`%a`	Locale's abbreviated weekday name
`%A`	Locale's full weekday name
`%b`	Locale's abbreviated month name
`%B`	Locale's full month name
`%c`	Locale's appropriate date and time representation
`%d`	Day of the month as a decimal number [01,31]
`%f`	Microsecond as a decimal number [0,999999], zero-padded on the left
`%H`	Hour (24-hour clock) as a decimal number [00,23]
`%I`	Hour (12-hour clock) as a decimal number [01,12]
`%j`	Day of the year as a decimal number [001,366]
`%m`	Month as a decimal number [01,12]
`%M`	Minute as a decimal number [00,59]
`%p`	Locale's equivalent of either AM or PM.
`%S`	Second as a decimal number [00,61]
`%U`	Week number (Sunday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Sunday are considered to be in week 0
`%w`	Weekday as a decimal number [0(Sunday),6]
`%W`	Week number (Monday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Monday are considered to be in week 0
`%x`	Locale's appropriate date representation
`%X`	Locale's appropriate time representation
`%y`	Year without century as a decimal number [00,99]
`%Y`	Year with century as a decimal number
`%z`	UTC offset in the form +HHMM or -HHMM (empty string if the the object is naive)
`%Z`	Time zone name (empty string if the object is naive)
`%%`	A literal `'%'` character

Logging exception information

- Use logging.exception()
- Adds exception info to message
- Only in **except** blocks

The `logging.exception()` method will add exception information to the log message. It may only be called in an **except** block.

Example

logging_exception.py

```
import logging

logging.basicConfig( # configure logging
    filename='../LOGS/exception.log',
    level=logging.WARNING,  # minimum level
)

for i in range(3):
    try:
        result = i/0
    except ZeroDivisionError:
        logging.exception('Logging with exception info') # add exception info to the log
```

../LOGS/exception.log

```
ERROR:root:Logging with exception info
Traceback (most recent call last):
  File "/Users/jstrick/curr/courses/python/common/examples/logging_exception.py", line
11, in <module>
    result = i/0
ZeroDivisionError: division by zero
ERROR:root:Logging with exception info
Traceback (most recent call last):
  File "/Users/jstrick/curr/courses/python/common/examples/logging_exception.py", line
11, in <module>
    result = i/0
ZeroDivisionError: division by zero
ERROR:root:Logging with exception info
Traceback (most recent call last):
  File "/Users/jstrick/curr/courses/python/common/examples/logging_exception.py", line
11, in <module>
    result = i/0
ZeroDivisionError: division by zero
```

Loggers

- Calls logging methods
- Default logger is root
- Configuration is inherited from root *

The default logger is called 'root'. If you call the various methods to send log messages directly from logging, such as `logging.warn()`, they are called from the root logger. Using `logging.basicConfig()` configures the root logger. If you set a minimum severity level on the root logger, it is used by all loggers.

To get the root logger, call `logging.getLogger()` with no parameters.

For simple cases, you can just use the root logger, but for larger projects, you will probably want to create specific loggers, and configure each one as needed.

Each logger can have its own configuration for destination (file or other), severity level, message format, and other attributes.

Sometimes you might want to use more than one logger, to make it easy to filter out messages in different parts of your application.

You can call `logging.getLogger()` with any name to get a logger with that name. Then you can call the logging methods from that logger. You can create other loggers by calling `logging.getLogger()` with a name. A logger is really just a name, used for configuration purposes. `logging.getLogger()` returns a logger object from which you can call the message methods.

A logger is essentially just a name, so using multiple loggers lets you add distinctive information to the log message without changing anything else. You can use the module's name plus a prefix or suffix.

Loggers are hierarchical, and the root logger is the implicit top of the hierarchy: the logger `spam.ham.toast` is really `root.spam.ham.toast`. Loggers inherit their configuration from root, and any parent loggers.

Example

logging_loggers.py

```
import logging

logging.basicConfig(
    format='%(levelname)s %(name)s %(asctime)s %(filename)s %(lineno)d %(message)s', #
format log entry
    datefmt="%Y:%M:%S::%X",  # set date/time format
    filename='../LOGS/loggers.log',
    level=logging.INFO,
)

logger_one = logging.getLogger(__name__ + "-1") # unique logger names
logger_two = logging.getLogger(__name__ + "-2")

logger_one.info("this is information")  # use logger 1
logger_two.warning("this is a warning")  # use logger 2
logger_one.info("this is information")
logger_two.critical("this is critical")
```

../LOGS/loggers.log

```
INFO __main__-1 2023:01:53::10:01:53 logging_loggers.py 14 this is information
WARNING __main__-2 2023:01:53::10:01:53 logging_loggers.py 15 this is a warning
INFO __main__-1 2023:01:53::10:01:53 logging_loggers.py 16 this is information
CRITICAL __main__-2 2023:01:53::10:01:53 logging_loggers.py 17 this is critical
```

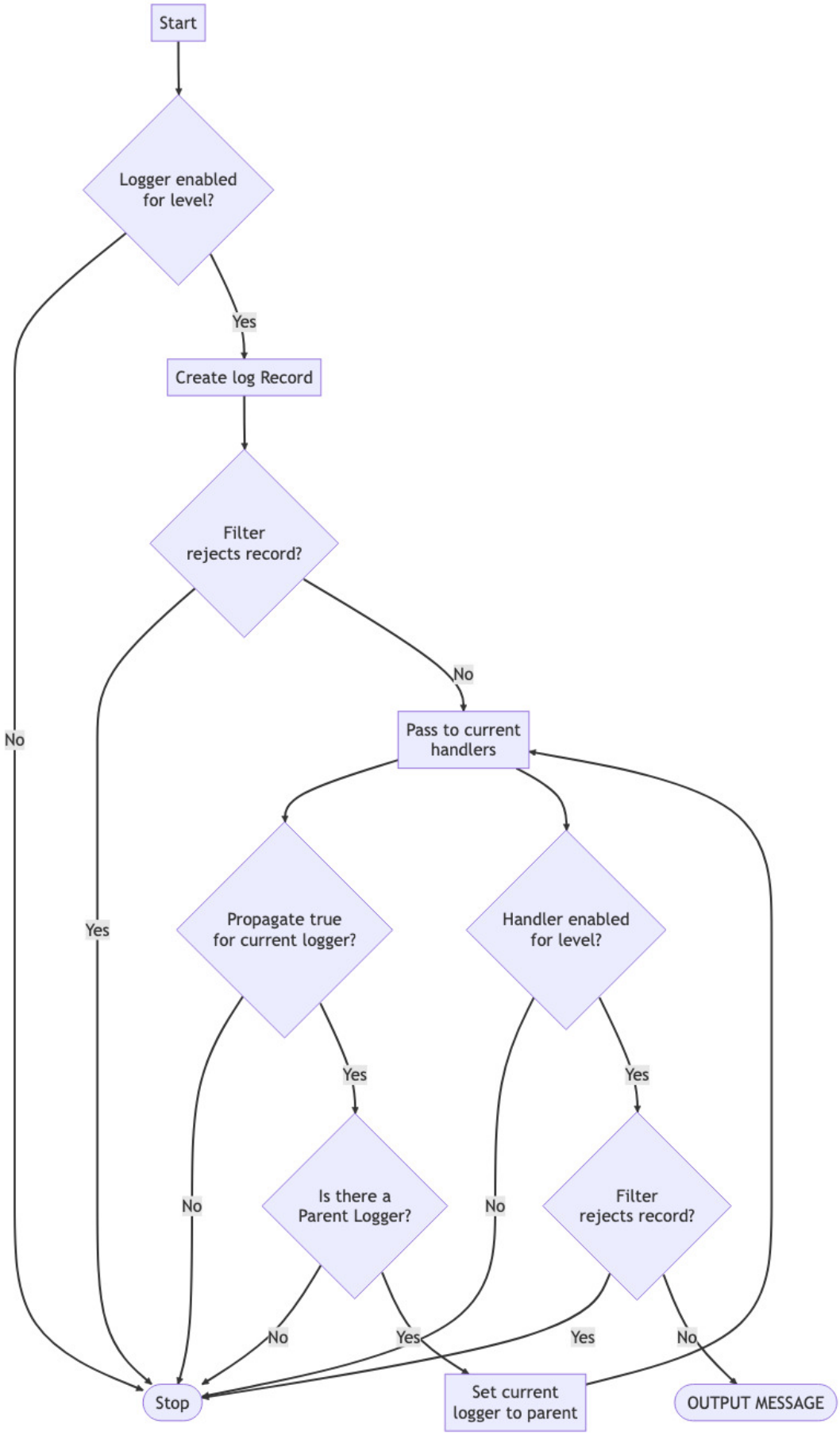

Figure 1. Logging Flowchart

Logging from modules

- No configuration in module
- Create module-specific logger

When you create a module, it should be independent of any code that uses the module. This applies to logging as well.

To keep a module independent, create a logger with a distinctive name. Best practice is to use the builtin `__name__` variable. This will make the logger the same name as the module.

Loggers in modules inherit the root logger configuration.

Example

apple.py

```
"""
Demonstrate logging from modules
"""
import logging
logger = logging.getLogger(__name__)


def apple():
    logger.warning("ambergris")
    logger.info("arrivederci")
    logger.error("aspiration")
    logger.debug("Function apple()")
```

banana.py

```
"""
Demonstrate logging from modules
"""
import logging
logger = logging.getLogger(__name__)

def banana():
    logger.warning("bravo")
    logger.info("Barbarella")
    logger.error("broccoli")
    logger.debug("banana()")
```

cherry.py

```
"""
Demonstrate logging from modules
"""
import logging
logger = logging.getLogger(__name__)

def cherry():
    logger.warning("ciao bello")
    logger.info("Cartesian")
    logger.error("cobblestones")
    logger.debug("Function cherry()")
```

logging_modules.py

```
import logging

logging.basicConfig(level=logging.DEBUG, filename="../LOGS/modules.log")

logger = logging.getLogger(__name__)

from apple import apple
from banana import banana
from cherry import cherry

def main():
    logger.info("Starting main script")
    apple()
    banana()
    cherry()
    logger.info("Ending main script")

if __name__ == '__main__':
    main()
```

../LOGS/modules.log

```
INFO:__main__:Starting main script
WARNING:apple:ambergris
INFO:apple:arrivederci
ERROR:apple:aspiration
DEBUG:apple:Function apple()
WARNING:banana:bravo
INFO:banana:Barbarella
ERROR:banana:broccoli
DEBUG:banana:banana()
WARNING:cherry:ciao bello
INFO:cherry:Cartesian
ERROR:cherry:cobblestones
DEBUG:cherry:Function cherry()
INFO:__main__:Ending main script
```

Multiple handlers

- Loggers can have one or more handlers
- Common handlers
  - `logging.FileHandler`
  - `logging.StreamHandler`

Every logger can use any number of handlers. Many handlers are provided by `logging` and `logging.handlers`.

The `logging` module provides many log handlers. If you specify `filename=` in `logging.basicConfig()`, it uses `logging.FileHandler`; if you specify `stream=`, it uses `logging.StreamHandler`.

TIP If using `logging.basicConfig()`, you can assign an iterable of one or more handlers with the `handlers=` parameter.

Example

logging_multi_handlers.py

```
import logging

screen_handler = logging.StreamHandler()
screen_handler.setLevel(logging.ERROR)

file_handler = logging.FileHandler("../LOGS/multihandlers.log")
file_handler.setLevel(logging.INFO)

logging.basicConfig(
    handlers=[screen_handler, file_handler],
)

logging.info("starting multi handler demo")
logging.warning("this is a warning")
logging.error("I lost my hat")
logging.critical("we are out of coffee")
logging.info("ending multi handler demo")
```

../LOGS/multihandlers.log

```
WARNING:root:this is a warning
ERROR:root:I lost my hat
CRITICAL:root:we are out of coffee
```

Beyond console and file logging

- Use specialized handlers to write to other destinations
- Multiple handlers can be added to one logger
  - `NTEventLogHandler` Windows event log
  - `SysLogHandler` **syslog**
  - `SMTPHandler` log to email
  - `NullHandler` suppress messages

The `logging` module provides many preconfigured log handlers to send log messages to destinations other than a file or the console. Each handler has custom configuration appropriate to the destination.

Multiple handlers can be added to the same logger, so a log message could go to both a file and to email. Each handler can have its own minimum severity level. Thus, all messages could go to the message file, but only CRITICAL messages would go to email.

Be sure to read the documentation for the particular log handler you want to use

TIP https://docs.python.org/3/howto/logging-cookbook.html has many recipes for customizing logging.

NOTE On Windows, `altdest.py` requires elevated privileges. To do this, launch the windows Command Prompt or the Anaconda prompt as administrator. m

Table 5. Handlers included in `logging` or `logging.handlers`

Handler	Description
`StreamHandler`	send messages to streams or other *opened* file-like objects.
`FileHandler`	send messages to files.
`BaseRotatingHandler`	base class for handlers that rotate log files. Should not to be instantiated directly.
`RotatingFileHandler`	same as `FileHandler`, but supports maximum log file size and log file rotation.
`TimedRotatingFileHandler`	same as `FileHandler`, but rotates files at specified intervals
`SocketHandler`	send messages to TCP/IP sockets. [1]
`DatagramHandler`	send messages to UDP sockets. [1]
`SMTPHandler`	send messages to specified email address.
`SysLogHandler`	send messages to Unix local or remote syslog (Not supported on Windows)
`NTEventLogHandler`	send messages to Windows event log. (Windows only)
`MemoryHandler`	send messages to memory buffer (flushed on specific criteria).
`HTTPHandler`	send messages to HTTP server using GET or POST.
`WatchedFileHandler`	watch log file; when file changes, close and reopen file. (Not supported on Windows)
`QueueHandler`	send messages to queue (from queue or multiprocessing modules)
`NullHandler`	do nothing with error messages; used by library developers who want to avoid 'No handlers could be found for logger XXX' messages

[1] Unix domain sockets supported since 3.4

Example

logging_altdest.py

```
import sys
from getpass import getpass
import logging
import logging.handlers

logger = logging.getLogger()  # get logger
logger.setLevel(logging.DEBUG)  # minimum log level

if sys.platform == 'win32':
    eventlog_handler = logging.handlers.NTEventLogHandler("Python Log Test")  # create NT
event log handler
    logger.addHandler(eventlog_handler)  # install NT event handler
else:
    syslog_handler = logging.handlers.SysLogHandler()  # create syslog handler
    logger.addHandler(syslog_handler)  # install syslog handler

smtp_password = getpass("SMTP Password: ")

# note -- use your own SMTP server...
email_handler = logging.handlers.SMTPHandler(
    ("smtp2go.com", 2525),
    'LOGGER@pythonclass.com',
    ['jstrickler@gmail.com'],
    'Alternate Destination Log Demo',
    ('pythonclass', smtp_password),
)  # create email handler

logger.addHandler(email_handler)  # install email handler

logger.debug('this is debug')  # goes to all handlers
logger.critical('this is critical')  # goes to all handlers
logger.warning('this is a warning')  # goes to all handlers
```

(no output to file or console)

Example

logging_null.py

```
import logging

logging.basicConfig(
    # filename='../LOGS/simple.log',
    level=logging.WARNING,
    handlers=[logging.NullHandler()],
)

logging.warning('This is a warning') # message will be output
logging.debug('This message is for debugging') # message will NOT be output
logging.error('This is an ERROR') # message will be output
logging.critical('This is ***CRITICAL***') # message will be output
logging.info('The capital of North Dakota is Bismark') # message will not be output
```

(no output to file or console)

Configuration

- Many approaches
  - `logging.basicConfig()`
  - `logging.dictConfig()`
  - `logging.fileConfig()`
  - *Ad hoc* configuration

In addition to `logging.basicConfig()`, you can configure logging via a dictionary or file.

Dictionary-based configuration

For dictionary-based configuration, use `logging.dictConfig()`, and pass in a dictionary whose keys are the same configuration options as `logging.basicConfig()`. There are a few differences. For severity levels, use `"ERROR"` instead of `logging.ERROR`. You can configure each logger by its name. The configuration version must be specified. This refers to the `logging` dictionary configuration schema, and is currently set to 1. In the future, if the schema should change, this will provide backwards compatibility.

This is considered best practice for logging configuration. If you want to keep the configuration in a file, you can store the configuration in a YAML or JSON file and read it into a Python dictionary.

NOTE `logging_config_yaml.py` and `logging_config_json.py` in the EXAMPLES folder show how to load a dictionary from a file.

Ad hoc configuration

You can also update logger and handler attributes on-the-fly.

Example

logging_config_dict.py

```
from argparse import ArgumentParser
import logging
from logging.config import dictConfig

# set up -d option to script
parser = ArgumentParser(description="Logging config demo")
parser.add_argument("-d", dest="debug", action="store_true", help="show debug messages")
args = parser.parse_args()

CONFIG = {
    "version": 1,  # required
    "loggers": {
        "root": {  # root can also be directly configured
            "level": 'ERROR',
            "handlers": ["console", "file"],  # handler IDs from this file
        },
    },
    "formatters": {
        "minimal": {
            "format": '%(name)s %(message)s'
        },
        "full": {
            "format": '%(asctime)s %(levelname)-10s %(name)-10s %(message)s',
            "datefmt": "%x %X",
        },
    },
    "handlers": {
        "console": {
            "class": "logging.StreamHandler",  # handler must be a string
            "formatter": "minimal",  # formatter ID from this file
            "level": "ERROR",    # translated to logging.ERROR
            "stream": "ext://sys.stderr",  # default is sys.stderr
        },
        "file": {
            "class": "logging.FileHandler",
            "formatter": "full",
            "filename": "../LOGS/dictconfig.log",
            "level": "DEBUG",
        },
    },
}

dictConfig(CONFIG)
```

```
if args.debug:
    root = logging.getLogger()
    # print(root.handlers)
    root.setLevel(logging.DEBUG)  # change level of file handler

logging.info("script begins")
logging.debug("configuration from a dictionary")
logging.error("Whoa -- this is not good")
logging.critical("Dude, we're doomed")
logging.info("script ends")
```

logging_config_dict.py

```
root Whoa -- this is not good
root Dude, we're doomed
```

../LOGS/dictconfig.log

```
04/19/23 10:01:53 ERROR      root       Whoa -- this is not good
04/19/23 10:01:53 CRITICAL   root       Dude, we're doomed
```

logging_config_dict.py -d

```
root Whoa -- this is not good
root Dude, we're doomed
```

../LOGS/dictconfig.log

```
04/19/23 10:01:53 ERROR      root       Whoa -- this is not good
04/19/23 10:01:53 CRITICAL   root       Dude, we're doomed
04/19/23 10:01:53 INFO       root       script begins
04/19/23 10:01:53 DEBUG      root       configuration from a dictionary
04/19/23 10:01:53 ERROR      root       Whoa -- this is not good
04/19/23 10:01:53 CRITICAL   root       Dude, we're doomed
04/19/23 10:01:53 INFO       root       script ends
```

File-based configuration

- Older method
- Not as flexible as dict config
- Quirks and disadvantages

To use configuration from a file, the format is similar to Windows `.INI` files. File configuration has been around longer than dictionary configuration, and there are some quirks and disadvantages, without any real advantages.

The entries in the file correspond to the `logging.basicConfig()` parameters and the keys used with `logging.dictConfig()`. The root logger must be configured with the `[logger_root]` entry.

In general, the best way to configure Python logging for non-trivial cases is to put the configuration information in a JSON or YAML file and read it into a dictionary, which is then passed to `logging.dictConfig()`.

Disadvantages

- No comments are allowed in the file
- Keys must be defined and then joined to "object type" names
- No indentation of nested information

Example

logging.conf

```
[loggers]
keys=root

[logger_root]
handlers=console,file
level=ERROR

[formatters]
keys=minimal,full

[formatter_minimal]
format=%(name)s %(message)s

[formatter_full]
format=%(asctime)s %(levelname)-10s %(name)-10s %(message)s
datefmt=%x %X

[handlers]
keys=console,file

[handler_console]
class=StreamHandler
formatter=minimal
level=ERROR
stream=ext://sys.stderr

[handler_file]
class=FileHandler
formatter=full
args=('../LOGS/fileconfig.log',)
level=INFO
```

logging_config_file.py

```
from argparse import ArgumentParser
import logging
from logging.config import fileConfig

# set up -d option to script
parser = ArgumentParser(description="Logging config demo")
parser.add_argument("-d", dest="debug", action="store_true", help="show debug messages")
args = parser.parse_args()

fileConfig('logging.conf')

if args.debug:
    root = logging.getLogger()
    # print(root.handlers)
    root.setLevel(logging.DEBUG)  # change level of file handler

logging.info("script begins")
logging.debug("configuration from a dictionary")
logging.error("Whoa -- this is not good")
logging.critical("Dude, we're doomed")
logging.info("script ends")
```

logging_config_file.py

```
root Whoa -- this is not good
root Dude, we're doomed
```

../LOGS/fileconfig.log

```
04/19/23 10:01:53 ERROR      root       Whoa -- this is not good
04/19/23 10:01:53 CRITICAL   root       Dude, we're doomed
```

logging_config_file.py -d

```
root Whoa -- this is not good
root Dude, we're doomed
```

../LOGS/fileconfig.log

```
04/19/23 10:01:53 ERROR      root       Whoa -- this is not good
04/19/23 10:01:53 CRITICAL   root       Dude, we're doomed
04/19/23 10:01:53 INFO       root       script begins
04/19/23 10:01:53 ERROR      root       Whoa -- this is not good
04/19/23 10:01:53 CRITICAL   root       Dude, we're doomed
04/19/23 10:01:53 INFO       root       script ends
```

Chapter 1 Exercises

Exercise 1-1 (president_log.py, pres_log_main.py)

Modify the existing module `president.py` to add logging. Add a critical message if the `presidents.txt` file is not found, and a warning message if an invalid term number is passed to the constructor. This should be done with `try-except` blocks.

Write a separate script to import the module and create one or more President instances. Try to create an instance with an invalid term number. Your module should log the error. Try temporarily changing the data file script name and see if you module correctly logs the error. The script should log when it starts and stops at `logging.INFO` level.

Index