logging_tutorial

February 15, 2018

1 Python Tutorial: Logging Basics - Logging to Files, Setting Levels, and Formatting

The code from the video.

1.1 How to start logging:

- Import `logging` (it's in the Standard Python Library)
- Add log statments to your code:

```
logging.debug('some print statement {}'.format(some_result_from_a_Function))
# or if you only want output to be logged above a certain logging level (see below for more in
logging.warning('some print statement {}'.format(some_result_from_a_Function))
```

5 Standard Logging Levels

- `DEBUG`: Detailed information, typically of interest only when diagnosing problems.
- `INFO`: Confirmation that things are working as expected.
- `WARNING`: An indication that something unexpected happened, or indicative of some problem in the near future (e.g. 'disk space low'). The software is still working as expected.
- `ERROR`: Due to a more serious problem, the software has not been able to perform some function.
- `CRITICAL`: A serious error, indicating that the program itself may be unable to continue running.
- Default level for logging is the Warning Level, which means everything from Warning to Critical will be logged, but Debug and Info will not be.

Changing the logging Level/Configuration:

- `python logging.basicConfig(level=logging.DEBUG)`
- `python logging.basicConfig(level=logging.INFO)`

Logging to the console:

- Done by default, if you do specify a logfile in logging.basicConfig
- "'python logging.debug('some print statement {}'.format(some_result))

Attribute name	Format	Description
args	You shouldn't need to format this yourself.	The tuple of arguments merged into `msg` to produce `message`, or a dict whose values are used for the merge (when there is only one argument, and it is a dictionary).
asctime	`%(asctime)s`	Human-readable time when the `LogRecord` was created. By default this is of the form '2003-07-08 16:49:45,896' (the numbers after the comma are millisecond portion of the time).
created	`%(created)f`	Time when the `LogRecord` was created (as returned by `time.time()`).
exc_info	You shouldn't need to format this yourself.	Exception tuple (à la `sys.exc_info`) or, if no exception has occurred, `None`.
filename	`%(filename)s`	Filename portion of `pathname`.
funcName	`%(funcName)s`	Name of function containing the logging call.
levelname	`%(levelname)s`	Text logging level for the message (`'DEBUG'`, `'INFO'`, `'WARNING'`, `'ERROR'`, `'CRITICAL'`).
levelno	`%(levelno)s`	Numeric logging level for the message (`DEBUG`, `INFO`, `WARNING`, `ERROR`, `CRITICAL`).
lineno	`%(lineno)d`	Source line number where the logging call was issued (if available).
module	`%(module)s`	Module (name portion of `filename`).
msecs	`%(msecs)d`	Millisecond portion of the time when the `LogRecord` was created.
message	`%(message)s`	The logged message, computed as `msg % args`. This is set when `Formatter.format()` is invoked.
msg	You shouldn't need to format this yourself.	The format string passed in the original logging call. Merged with `args` to produce `message`, or an arbitrary object (see Using arbitrary objects as messages).
name	`%(name)s`	Name of the logger used to log the call.
pathname	`%(pathname)s`	Full pathname of the source file where the logging call was issued (if available).
process	`%(process)d`	Process ID (if available).
processName	`%(processName)s`	Process name (if available).
relativeCreated	`%(relativeCreated)d`	Time in milliseconds when the LogRecord was created, relative to the time the logging module was loaded.
stack_info	You shouldn't need to format this yourself.	Stack frame information (where available) from the bottom of the stack in the current thread, up to and including the stack frame of the logging call which resulted in the creation of this record.
thread	`%(thread)d`	Thread ID (if available).
threadName	`%(threadName)s`	Thread name (if available).

16_6_7_LogRecord_attributes.png

2 Output:

DEBUG::<: # You could also manually format it like so: DEBUG::<function_name: your_print_statement>"'

Using a Log File:

- `python logging.basicConfig(filename='test.log', level=logging.INFO) # include these as well logging.debug('some print statement {}'.format(some_result))`
- Log files are saved in the current working directory by default.

Changing the format of your log statements:

- Possible Format options can be found in the table below.
- `logging.basicConfig(filename='test.log', level=logging.INFO, format="<attribute_1>:<attribute_2>:.....')`

```
# Example (doesn't neccessarly need the colons):
logging.basicConfig(filename='test.log', level=logging.INFO,
format='%(asctime)s:%(levelname)s:%(message)s')
```

2.1 Advanced Logging - Custom, Named loggers.

NOTE - When importing modules that use only the basic logging configuaraton method '`logging.basicConfig()`' into a python file that already has its own basic logging configuaraton method; a problem may arise if you're not careful. * For instance, if not specifying a unique

logger for each file, only the imported module may run its log statements. It all depends where the imported module has set its log level, and what the individual log statements in the non-imported file have their levels set to. * For example, if the imported module has its level set to warning, and the statments in the non-imported module have theirs set to debug, they'll never be logged. * To Remedy this, use the following:

```
<logger_name> = logging.getLogger(__name__) # Doesn't not have to be Dunder name, but it is co
<logger_name>.setLevel(logging.<level>)  # Level could be DEBUG, WARNING, INFO, etc.

<formatter_name> = logging.Formatter(<format_of_choice>) # Format could be something like '%(a

<fileHandler_name> = logging.FileHandler(<name_of_log_file>) # Could be something like 'employ
# Optional ------------------------------------------------------------------------------------
# Let say after you're done debugging, you now only want to save your log statments that are a
<fileHandler_name>.setLevel(logging.<level>)
# ------------------------------------------------------------------------------------------------
<fileHandler_name>.setFormatter(<formatter_name>)

<logger_name>.addHandler(<fileHandler_name>)

# BE SURE ALL LOGGING STATMENTS HAVE THEIR LOGGER'S NAME:
<logger_name>.debug(........)
```

- FYI: Typically the default logger is the `root` logger, which is why you see `root` in the logger's output above.

Logging with `Try & Except` Blocks

- To include a traceback when logging use `<logger_name>.exception(.....)`
- To not include traceback just use the `<level>` you want.

```
def divide(x, y):
    """Divide Function"""
    try:
        result = x / y
    except ZeroDivisionError:
        <logger_name>.exception('Tried to divide by zero')
        #  OR  <logger_name>.ERROR('Tried to divide by zero')
    else:
        return result
```

Logging DEBUG at the console, and logging other `<levels>` with a file On top of what was written above in Advanced Logging, you can include the following:

```
# SreamHandler's LOGGING LEVEL IS BASED ON LOGGING LEVEL SET UP above: <logger_name>.setLevel(
<streamHandler_name> = logging.StreamHandler()
<streamHandler_name>.setFormatter(<formatter_name>)  # You could create a custom format, diffe

<logger_name>.addHandler(<streamHandler_name>)
```