# Application Level Logging best practices

* Author: Usama Altaf

## Logging details

**Logging is the process of recording application actions and state to a secondary interface.**

This user-interface is the primary output mechanism that the software uses to communicate with the outside world. In Use-Case-speak, it is the system's mechanism for communicating with the primary actor. It can be advantageous to the process of maintaining an application to have some record of the actions it performs and changes to its internal state. This information is usually of no concern to the primary user; therefore, the application requires a secondary interface for communication.

# 

## What Do Application Logs Contain?

Each application you use contains code to write different types of events to an application log file. Events within an application log file can vary, but typically include:

1. Application exceptions

2. Major events like startups, stops, restarts, and security events

3. Error event notifications

4. Debug information

5. SQL logs

6. Warnings about low disk space

These events are classified as error, warning, or information, depending on the severity of the event, and often come equipped with a time stamp. With application log management software, you can set alert thresholds and receive instant updates if an event has crossed your predetermined threshold, making it easier to navigate the volumes of event data. These application logging tools even let you know the problem area within the application, so you can take a swift course of action.

## Logging messages

### DEBUG

This log level is used to indicate that the logged message is to be used for debugging purposes - in other words, these messages are aimed squarely at the developer. What you use this for really depends on the application you are developing. Many problems can be resolved via the debugger, making the use of DEBUG messages redundant; however, there are a few situations where DEBUG messages are quite useful, for example:

* In debugging graphics rendering desktop applications, the process of firing up the debugger may interfere with the render process itself. In these instances, DEBUG messages can be a useful way of 'watching' the render process in a less obtrusive fashion.
* Some application types or stages in the development process require special environments for application execution. In these cases, using a debugger tool may not be a viable option. In these cases, DEBUG messages can be a useful instrument for finding the solution to tricky bugs.

The above are just two examples where DEBUG messages are useful. As stated before, the usefulness really does depend on the nature of the application being developed.

### 

### INFO

* As the name implies, these messages are purely for sharing information. They should not be used to signal a problem or error in the application. To make the most of this log level, think about what general information would be helpful for diagnosing application issues when you can't access the main interface. Some useful information could include:
* Software version details: It's a good practice to log the software version when the application starts. This helps avoid confusion when troubleshooting problems.
* Usage information: Who is using the software, and what are they doing? This can provide context for understanding user interactions.
* External services: Log information about the external services the application relies on, like databases or web services. This can be crucial for identifying issues related to these dependencies.

### WARN

This is the first level which indicates some form of application failure. With three levels at your disposal, consistent usage guidelines are important. WARN level messages should be used to indicate that the application faced a potential problem; however, the user experience has not been affected in any way. For example, a WARN message might be appropriate if an external service could not be used; however, a secondary service which performs the same functions was available. Also, a WARN message is appropriate if repeated attempts were required to access a given resource.

### ERROR

This is the second level of failure, and by its very name, it should indicate that something more critical has occurred. ERROR messages should be used to indicate that the application faced a significant problem and that, as a result, the user experience was affected in some way. For example, a database connection could have failed, resulting in parts of the application being rendered unusable.

### 

CRITICAL

This third level of failure should be used to indicate a fatal error. The user experience was not just affected, it has entirely ceased! For example, a component which is central to the operation of the application has failed in a way that leaves it in an unstable state, with the only possible course of action being to terminate the application altogether.

## Points to remember:

DEBUG: Detailed information, for diagnosing problems. Value=10.

INFO: Confirm things are working as expected. Value=20.

WARNING: Something unexpected happened, or indicative of some problem. But the software is still working as expected. Value=30.

ERROR: More serious problem, the software is not able to perform some function. Value=40

CRITICAL: A serious error, the program itself may be unable to continue running. Value=50

## 

## Difference between Logging Debug vs Info:

| **Aspect** | **Debug Logging** | **Info Logging** |
| --- | --- | --- |
| **Purpose** | To assist developers in diagnosing issues. | To provide general operational information. |
| **Detail Level** | Very high, with granular details. | Moderate, focusing on significant events. |
| **Use Case** | Primarily in development or for troubleshooting. | Suitable for both development and production. |
| **Content** | In-depth details like variable values, system states. | High-level information like start/end of operations. |
| **Volume** | High, can generate lots of logs. | Lower, only logs key events. |
| **Performance Impact** | Can slow down the system due to volume and detail. | Minimal impact, as the log volume is lower. |

| **Scenario** | **Debug Logging Example** | **Info Logging Example** |
| --- | --- | --- |
| **User Login Attempt** | "Login attempt for user\_id=12345 from IP: 192.168.1.1; input length=20 characters; timestamp=08:00:03 UTC" | "User login successful for user\_id=12345 at 08:00:03 UTC." |
| **Payment Processing** | "Initiating payment: amount=$150; user\_id=12345; currency=USD; calling external API /payment/process" | "Payment of $150 processed for user\_id=12345." |
| **File Upload** | "Start uploading file 'data.csv': size=3.5MB; split into 5 chunks; chunk 1 uploaded; timestamp=09:15:10 UTC" | "File 'data.csv' uploaded successfully at 09:16:00 UTC." |
| **System Resource Check** | "Server resource check: CPU usage=65%; Memory usage=70%; Disk space remaining=120GB; timestamp=10:00:00 UTC" | "System health check completed at 10:00:00 UTC. All systems operational." |
| **Error Handling** | "Error in processOrder(): OrderID=7890; ErrorType=DatabaseTimeout; RetryAttempt=1; timestamp=11:30:45 UTC" | "Temporary issue encountered processing order 7890. Resolved." |

# 

## Best way of logging:

## 

### Include important information:

Add context to the message content, such as:

* What action was performed
* Who performed the action
* Why a failure occurred
* Remediation information when possible for WARN and ERROR messages
* HTTP request ID - add Request ID for information

## 

### Use structured logging:

Human-readable logs are helpful, but structured logs have more value, especially when your application's logs get more complicated or when there are a lot of requests. Structured logs use a clear format for log entries, making it easier to organize, search, and analyze them.

JSON is the de facto standard for structured logging, but consider using key=value pairs, XML, or another format for your application logs. Review the supported formats that your add-on provider can automatically parse.

**Consider these unstructured log entries:**

2019-06-20T17:21:00.002899+00:00 app[email-wrkr.1]: DEBUG: Fetching mailing list 14777

2019-06-20T17:22:00.000098+00:00 app[email-wrkr.1]: DEBUG: User 3654 opted out

**Changing to more info:**

2019-06-20T17:21:00.002899+00:00 app[email-wrkr.1]: DEBUG: Fetching mailing list {"listid":14777}

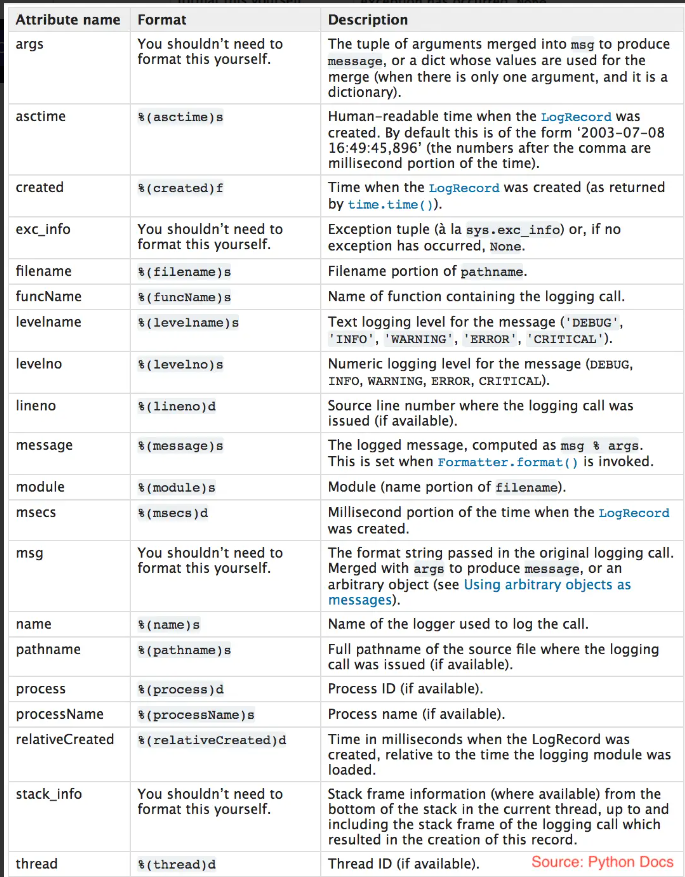
2019-06-20T17:22:00.000098+00:00 app[email-wrkr.1]: DEBUG: User opted out {"userid":3654}

**Logging formats can be changed using format as shown below**

import logging

logging.basicConfig(level=logging.INFO, format='%(asctime)s :: %(levelname)s :: %(message)s')

logging.info("Just like that!")



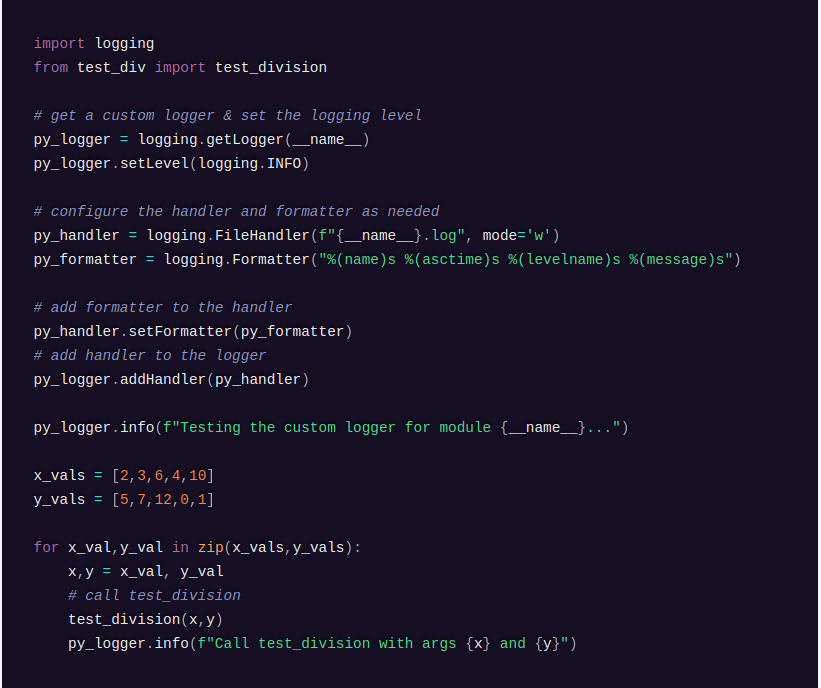
### Named Loggers:

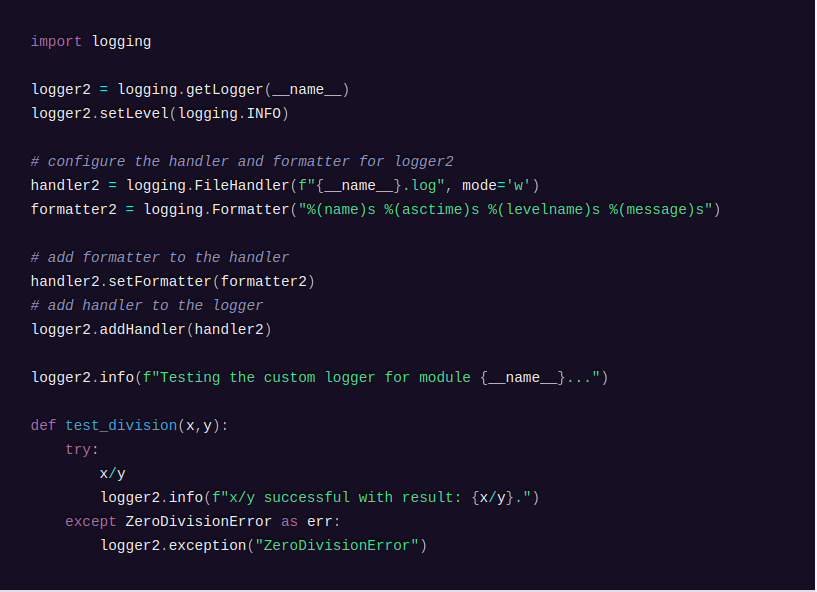
Named loggers are a helpful concept not only for specific components within an application but also for different parts of the application itself. You can use named loggers to distinguish log messages from various application areas, such as the database (DB), frontend, or any other significant module or service.

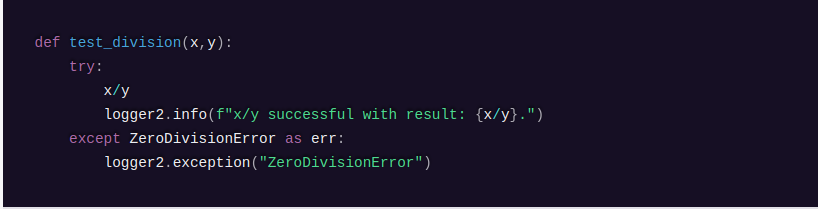
For instance, you can create named loggers like "DBLogger," "FrontendLogger," or "SecurityLogger" to separate and label log messages from different parts of your application. This naming convention allows you to easily identify and categorize log entries by their source, making it simpler to troubleshoot issues and track the behavior of specific application components. It's like having different folders to keep your files organized, but in this case, it's for your log messages.

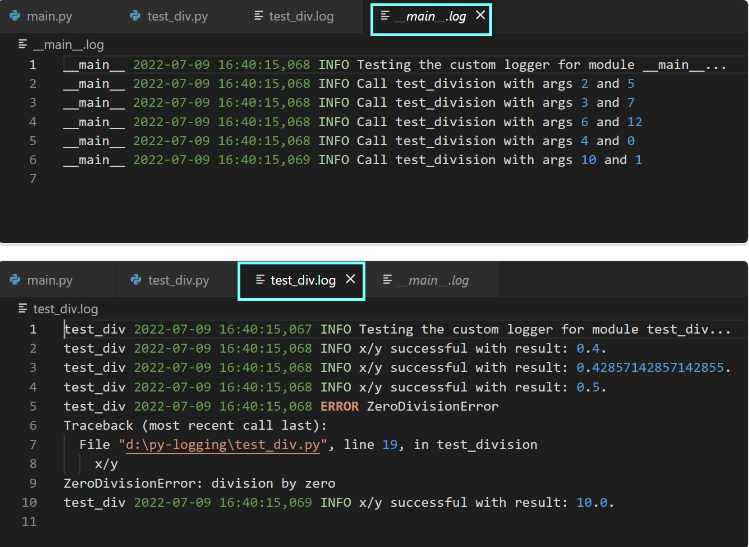
A logger is like a nametag. When you give it a name, it writes that name in the log every time it's used to put a message in the log. So, what name should we use? Well, our application code already has a way to organize things, like the class name and where it belongs. We can use this organization to name our loggers.

In the code for a part of our app called "Acme.Components.Things," let's say we have a class called "WidgetClass." We can create a logger for that class, and we'll name it "**Acme.Components.Things.WidgetClass**." That way, any messages written by this class's logger will show **"Acme.Components.Things.WidgetClass**" in the log. This helps us know exactly which part of our app is responsible for each message. It's like putting labels on things to keep them organized.

****

****

****

****

Let’s parse what the above code for configuring custom loggers does.

As a first step, we set up the logger and the logging level. logging.getLogger(name) returns the logger with that name, if it exists; else, it creates the name logger. In practice, you’ll set the name of the logger to the special variable \_\_name\_\_, which corresponds to the name of the module. We assign the logger object to a variable. We then set the desired logging level using logging.setLevel(level).

Next, we configure a handler. As we’d like to log events to a file, we configure a FileHandler. logging.FileHandler(filename) returns a file handler object. In addition to the name of the log file, you may optionally specify the mode. In this example, we set the mode to write. There are other handlers such as StreamHandler, HTTPHandler, SMTPHandler, and more.

We then create a formatter object using the syntax: logging.Formatter(format). In this example, we place %(names)s, the name of the logger (a string), before the log record format we had earlier.

Next, we add the formatter to the handler using <handler>.setFormatter(<formatter>). Finally, we add the handler to the logger object using <logger>.addHandler(<handler>).

You can then run the main module and examine the generated log files.

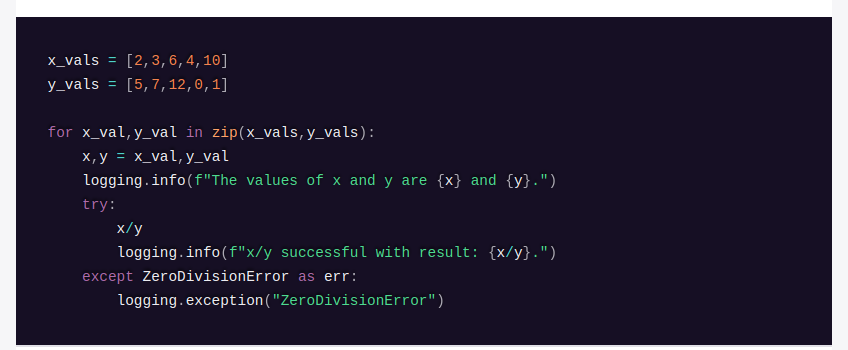
## 

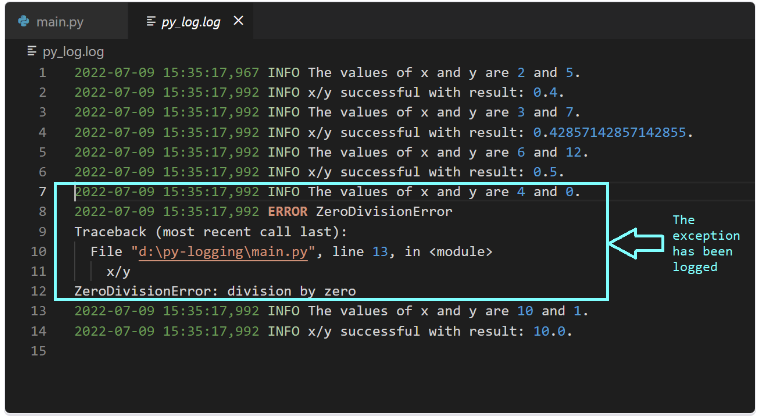
## 

### How to include traceback information in logged messages

Now, let’s modify the main.py file. Say, there are two variables x and y, and we’d like to compute the value of x/y. We know that we’ll run into ZeroDivisionError when y = 0. We can handle this as an exception using the try and except blocks.

Next, we’d like to log the exception along with the stack trace. To do this, you can use logging.error(message, exc\_info=True). Run the following code to see that the values of variables and the result are logged as INFO, indicating that the code works as expected.



****

Now in this example, as you can see, logging gives the exceptions but also traceback the line number on which error is coming.

## Examples

**Example 1:**

**Code at this** [**link**](https://github.com/Shivakoreddi/ETL-Log-Structure/blob/main/etl_log_job.py)

**Logging results:**

****

**Explanation:**

As you can see in the above example, each info which is required without the need of looking at code is being added using Info method, furthermore, the logging is named with class to make it easy to spot the functions or location using **Named Loggers**