

Logging in Python

Author: Shoaib Khan

What is logging ?

Logging is a means of tracking events that happen when some program runs. The developer adds logging calls to their code to indicate that certain events have occurred. An event is described by a descriptive message which can optionally contain variable data. In machine learning context, it could be names of **features** used in the model, model **hyperparameters**, **data location** etc. (<https://docs.python.org/3/howto/logging.html>)

Logging can later help to debug/resolve errors in the program. it allows to preserve/save warning, errors and general information as the code executes.

However, logging is very much controlled by the user, within them they mostly contain user defined messages. **The clearer the message the easier it is to debug.** but, The user can also allow system level logs and stack-trace messages to be logged as well.

When to use logging ?

Generally, logging is used when you are creating a set of modules/code files in a project. While you can use logging in Jupyter notebooks, it mostly depends on the complexity of the code. for e.g. we don't see logging implemented in simple EDAs (exploratory data analysis).

logging v/s print statements?

print statements are great for smaller applications but as mentioned above logging becomes an essential component as your simple EDA turns into a project and its final destination is in some production server. logging would allow the log messages to be saved and so you always have a history of code execution.

logging levels?

There are various logging levels that dictate the messages types that will be logged. By default, logging is set to **Warning** level and thus any log messages below it, won't not be logged. for e.g. if logging is set to **INFO** level then **DEBUG** level messages will not be logged but anything above it, will be.

Notset = 0	This is the initial default setting of a log when it is created. It is not really relevant and most developers will not even take notice of this category. In many circles, it has already become nonessential. The root log is usually created with level WARNING.
Debug = 10	This level gives detailed information, useful only when a problem is being diagnosed.
Info = 20	This is used to confirm that everything is working as it should.
Warning = 30	This level indicates that something unexpected has happened or some problem is about to happen in the near future.
Error = 40	As it implies, an error has occurred. The software was unable to perform some function.
Critical = 50	A serious error has occurred. The program itself may shut down or not be able to continue running properly.



logging — Logging facility for Python — Python 3.10.1
<https://docs.python.org/3/library/logging.html#logging-levels>

Basic Logging setup

1. A simple example for logging

```
#ex1.py

import logging
from sklearn.datasets import fetch_california_housing

logging.basicConfig() #by default logging level is at WARNING level
logging.warning("getting data from sklearn datasets")

data = fetch_california_housing()
```

```

🍏 ~ /engineering/Logging_example ..... ml_notebooks 15:40:30
> python3 ex1.py
WARNING:root:getting data from sklearn datasets
```

2. still simple with few bells

The first example simply printed the message, which is certainly isn't preserved. Thus, lets try to save the messages to a file and set a logging level. Further, if you look at the following documentation, it suggests that you can format your messages, such as add date-time and a lot of other useful components to your log messages.



logging — Logging facility for Python — Python 3.10.1

<https://docs.python.org/3/library/logging.html#logrecord-attributes>

you can define them using `filename`, `filemode`, `level` and `format` arguments.

```
#ex1.py

import logging
from sklearn.datasets import fetch_california_housing

#by default logging level is at WARNING level
logging.basicConfig(
    filename="ex1.log",
    filemode="w",
    level=logging.INFO,
    format="%(asctime)s: %(levelname)s: %(name)s: %(module)s: %(message)s",
)
logging.info("getting data from sklearn datasets")

data = fetch_california_housing()
```

Once we execute the above, we will notice that the log output is being saved in **training.log** file and the `filemode: w` suggest that the training.log file is overwritten every time we execute it. We could set this to `a` which indicates **append** and it is the default behaviour.

```
1 2022-01-09 20:47:35,512: INFO: root: ex1: getting data from sklearn datasets
```

3. let's bring in another module

The syntax in this example should now look familiar to you. We are logging an info level message by creating another logger in a different file called `ex2.py`.

Before executing the below `ex2.py`, our project structure looks like the following.

~/engineering/logging_example

> tree

```
.
├── ex1.log
├── ex1.py
└── ex2.py
```

0 directories, 3 files

```
#ex2.py

import logging

logging.basicConfig(
    filename="ex2.log",
    filemode="w",
    level=logging.INFO,
    format="%(asctime)s: %(levelname)s: %(name)s: %(module)s: %(message)s",
)

logging.info("This is a message from the future!!")
```

After executing ex2.py we have another logfile called ex2.log (notice we are saving ex2 log into a different file) and our project structure look is updated.

~/engineering/logging_example

> tree

```
.
├── ex1.log
├── ex1.py
├── ex2.log
└── ex2.py
```

0 directories, 4 files

The following is message logged and everything is working as expected.

```
ex2.py  ex2.log x
ex2.log
1 2022-01-09 16:53:26,846: INFO: root: ex2: This is a message from the future!!
```

4. bring in the two culprits

Let's clean up our project directory and start afresh with two .py files.

~/engineering/logging_example

> tree

```
.
├── ex1.py
└── ex2.py
```

0 directories, 2 files

At this point, we will do something interesting, i.e. we will import `ex2.py` in `ex1.py`. so the code in both files looks like below.

```
#ex1.py

import logging
from sklearn.datasets import fetch_california_housing

#by default logging level is at WARNING level
logging.basicConfig(
    filename="ex1.log",
    filemode="w",
    level=logging.INFO,
    format="%(asctime)s: %(levelname)s: %(name)s: %(module)s: %(message)s",
)
logging.info("getting data from sklearn datasets")

data = fetch_california_housing()
```

```
#ex2.py

import logging

logging.basicConfig(
    filename="ex2.log",
    filemode="w",
    level=logging.INFO,
    format="%(asctime)s: %(levelname)s: %(name)s: %(module)s: %(message)s",
)

logging.info("This is a message from the future!!")
```

we will now import `ex2` module in `ex1.py` and execute `ex1.py`. So now the our `ex1` script is updated to look like this.

```
#ex1.py

import logging
from sklearn.datasets import fetch_california_housing

import ex2

#by default logging level is at WARNING level
logging.basicConfig(
    filename="ex1.log",
    filemode="w",
    level=logging.INFO,
    format="%(asctime)s: %(levelname)s: %(name)s: %(module)s: %(message)s",
)
logging.info("getting data from sklearn datasets")

data = fetch_california_housing()
```

pause and think! what should be the output of this, should you expect ?

- 1 log file (which one ex1.log or ex2.log)
- 2 log files
- 0 log file
- error

Well before I mention the answer, know that, in python whenever we import a module everything in that module is executed before the file where the import is taking place.

Thus, as we imported ex2 module in ex1, ex2 script was executed first and this meant that the logging i.e `logging.basicConfig()` was setup by ex2 and not by ex1. So in this case the **root logger** was setup by ex2. Therefore, only 1 log file (ex2.log) gets created. However, as expected log messages from `ex1.py` `ex2.py` are logged.

```
2022-01-09 21:13:13,035: INFO: root: ex2: This is a message from the future!!
2022-01-09 21:13:13,621: INFO: root: ex1: getting data from sklearn datasets
```

Create new loggers in place of ROOT

Till now, we have seen that we were able to log files but it was dictated by the root logger. We want to create instances of logging i.e loggers for each module. for e.g. we may want to save our logs into different log files.

1. Start afresh

We would start afresh with our `ex1` and `ex2` files and rewrite them. At this point, we would simply work with `ex1.py` and later on copy most of the logging setup code into `ex2.py` along with a minor change.

So to remind ourselves, we go through some steps when setting up logging and they are,

- logging level
- filename, i.e. name of the log file.
- filemode, we either append to the log file or rewrite.
- formatter to set the format for the output.

We will go through same setup to instantiate a logging instance but in a different way. Here is the code, we will dissect it in a moment.

```
import ex2
import logging
from sklearn.datasets import fetch_california_housing

# a logging instance
logger = logging.getLogger(__name__)

# set a logging level
logger.setLevel(logging.INFO)

# formatting the log output
formatter = logging.Formatter(
    "%(asctime)s: %(levelname)s: %(name)s: %(module)s: %(message)s"
)

# filehandler that creates a log file
file_handler = logging.FileHandler(filename="ex1.log", mode="w")

# passing the expected log output format to the filehandler
file_handler.setFormatter(formatter)

# passing the handler to the logger
logger.addHandler(file_handler)

# use the instance to log messages
logger.info("getting data from sklearn datasets")
data = fetch_california_housing()
```

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

We start with setting up a logger instance and pass it with `__name__`, it represents the name of the module where the code lives.

```
logger.setLevel(logging.INFO)
```

This simply sets the logging level to the logging instance which we called logger.

```
formatter = logging.Formatter("%(asctime)s: %(levelname)s: %(name)s: %(module)s: %(message)s")
```

This is where we define the format of logging and various formats can be found [here](#).

```
file_handler = logging.FileHandler(filename="ex1.log", mode="w")
```

The file handler as the name suggest handles the file and takes in similar arguments such as `logging.basicConfig`

```
file_handler.setFormatter(formatter)
```

```
logger.addHandler(file_handler)
```

Lastly we set the formatter and add the file-handler to the logging instance called logger and we are ready to use logger to log.

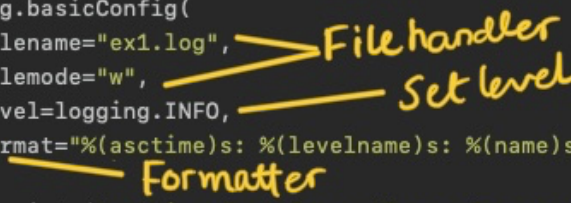
Below is an example of where some of the components will go in the steps described above.

```
#ex1.py

import logging
from sklearn.datasets import fetch_california_housing

#by default logging level is at WARNING level
logging.basicConfig(
    filename="ex1.log",
    filemode="w",
    level=logging.INFO,
    format="%(asctime)s: %(levelname)s: %(name)s: %(module)s: %(message)s",
)
logging.info("getting data from sklearn datasets")

data = fetch_california_housing()
```



Right at the end, as discussed we will copy most of the logging code from `ex1.py` over to `ex2.py` and the only required to change here is the filename, which we want to save to `ex2.log`.


```
# ex2.py

import logging

logger = logging.getLogger(__name__)
logger.setLevel(logging.INFO)

formatter = logging.Formatter(
    "%(asctime)s: %(levelname)s: %(name)s: %(module)s: %(message)s"
)
file_handler = logging.FileHandler(filename="ex2.log", mode="w")
file_handler.setFormatter(formatter)

logger.addHandler(file_handler)

logger.info("This is a message from the future!!")
```

2. log to console as well as file

Till now, we have logged to a log file which is a persistent as its saved in your project directory. However, may be there is a need to not only but also print to the console/terminal. This is where we can use `StreamHandler()` method from logging. We would need to add some bit to the code.

```
# ex1.py

import ex2
import logging
from sklearn.datasets import fetch_california_housing

logger = logging.getLogger(__name__)
logger.setLevel(logging.INFO)

formatter = logging.Formatter(
    "%(asctime)s: %(levelname)s: %(name)s: %(module)s: %(message)s"
)
file_handler = logging.FileHandler(filename="ex1.log", mode="w")
stream_handler = logging.StreamHandler()

file_handler.setFormatter(formatter)
stream_handler.setFormatter(formatter)

logger.addHandler(file_handler)
logger.addHandler(stream_handler)

logger.info("getting data from sklearn datasets")
data = fetch_california_housing()
```

The parts that were added above are highlighted below.

```
# ex1.py

import ex2
import logging
from sklearn.datasets import fetch_california_housing

logger = logging.getLogger(__name__)
logger.setLevel(logging.INFO)

formatter = logging.Formatter(
    "%(asctime)s: %(levelname)s: %(name)s: %(module)s: %(message)s"
)
file_handler = logging.FileHandler(filename="ex1.log", mode="w")
stream_handler = logging.StreamHandler()

file_handler.setFormatter(formatter)
stream_handler.setFormatter(formatter)

logger.addHandler(file_handler)
logger.addHandler(stream_handler)

logger.info("getting data from sklearn datasets")
data = fetch_california_housing()
```

We can add `StreamHandler()` to `ex2.py` as well and then finally execute `ex1.py`

```
# ex2.py

import logging

logger = logging.getLogger(__name__)
logger.setLevel(logging.INFO)

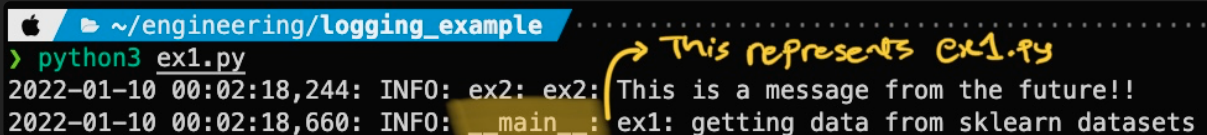
formatter = logging.Formatter(
    "%(asctime)s: %(levelname)s: %(name)s: %(module)s: %(message)s"
)
file_handler = logging.FileHandler(filename="ex2.log", mode="w")
stream_handler = logging.StreamHandler()

file_handler.setFormatter(formatter)
stream_handler.setFormatter(formatter)

logger.addHandler(file_handler)
logger.addHandler(stream_handler)

logger.info("This is a message from the future!!")
```

Now, not only are we saving logs but also printing them to the console.



```
Apple ~ /engineering/logging_example
> python3 ex1.py
2022-01-10 00:02:18,244: INFO: ex2: ex2: This is a message from the future!!
2022-01-10 00:02:18,660: INFO: __main__: ex1: getting data from sklearn datasets
```

This represents ex1.py

```
~/engineering/logging_example
```

```
> tree
```

```
.
├── ex1.log
├── ex1.py
├── ex2.log
└── ex2.py
```

```
0 directories, 4 files
```

log exceptions

logging exceptions are also extremely easy. We just need to use `logger.exception()` to log exceptions here is `ex3.py` demonstrating this.

```
# ex3.py

import logging

logger = logging.getLogger(__name__)
logger.setLevel(logging.INFO)

formatter = logging.Formatter(
    "%(asctime)s: %(levelname)s: %(name)s: %(module)s: %(message)s"
)
file_handler = logging.FileHandler(filename="ex3.log", mode="w")
stream_handler = logging.StreamHandler()

file_handler.setFormatter(formatter)
stream_handler.setFormatter(formatter)

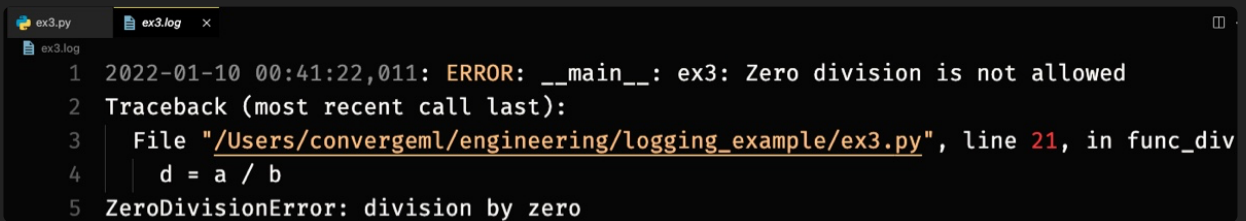
logger.addHandler(file_handler)
logger.addHandler(stream_handler)

def func_div(a, b):
    try:
        c = a / b
    except ZeroDivisionError:
        logger.exception("Zero division is not allowed")
    else:
        return c

if __name__ == "__main__":
    a = 0
    b = 3

    func_div(b, a)
```

when we execute the file `ex3.py` , our exception is logged with level ERROR.



```
ex3.py  ex3.log x
ex3.log
1 2022-01-10 00:41:22,011: ERROR: __main__: ex3: Zero division is not allowed
2 Traceback (most recent call last):
3   File "/Users/convergeml/engineering/logging_example/ex3.py", line 21, in func_div
4     d = a / b
5 ZeroDivisionError: division by zero
```

log files based on size

If we wanted to save our logs based on the size and break log files into separate smaller files use `RotatingFileHandler` and avoid crashing your system as you try to open a very large log file.

For more details, check here.



logging.handlers — Logging handlers — Python 3.10.1

<https://docs.python.org/3/library/logging.handlers.html#rotatingfilehandler>

The code replaces the `FileHandler()` with `RotatingFileHandler()` and the arguments `maxBytes` determines how the large each log file can be before logging is passed to another log file in sequence and `backupCount` determines how many log files it should break into. for e.g. if you state `backupCount` as 10 then we might end up with 11 files if all files are needed for logging.

Try running the code by changing the range to 1000.

```
# ex4.py

import logging
from logging.handlers import RotatingFileHandler

logger = logging.getLogger(__name__)
logger.setLevel(logging.INFO)

formatter = logging.Formatter(
    "%(asctime)s: %(levelname)s: %(name)s: %(module)s: %(message)s"
)

rotating_handler = RotatingFileHandler(
    filename="ex4.log", maxBytes=3000, backupCount=10
)

stream_handler = logging.StreamHandler()

rotating_handler.setFormatter(formatter)
stream_handler.setFormatter(formatter)

logger.addHandler(rotating_handler)
logger.addHandler(stream_handler)

for i in range(100):
    logger.info(f"this is message # {i}")
```

if I run this with just 100, I am presented with an output on the console since we have `StreamHandler()` but also get 3 logs files for `ex4.py` while we had kept 10 log files as a backup.

Apple ~/engineering/logging_example

> tree

```
.
├── ex1.py
├── ex2.py
├── ex3.py
├── ex4.log
├── ex4.log.1
├── ex4.log.2
└── ex4.py
```

0 directories, 7 files

log files based on time

If the need is to create a separate log file every 5 minutes or 5 seconds `TimedRotatingFileHandler()` is the method to use. Along with a familiar `backupCount` argument, it needs `when` to log i.e. log in seconds, minutes etc.. and `interval` which indicates do we need to log every 5 seconds or 5 minutes etc.

For more details, check [here](#).



We are logging here every second for 6 seconds.

```
# ex5.py

import logging
import time
from logging.handlers import TimedRotatingFileHandler

logger = logging.getLogger(__name__)
logger.setLevel(logging.INFO)

formatter = logging.Formatter(
    "%(asctime)s: %(levelname)s: %(name)s: %(module)s: %(message)s"
)
timed_handler = TimedRotatingFileHandler(
    filename="ex5.log", when="s", interval=1, backupCount=10
)
stream_handler = logging.StreamHandler()

timed_handler.setFormatter(formatter)
stream_handler.setFormatter(formatter)

logger.addHandler(timed_handler)
logger.addHandler(stream_handler)

for i in range(6):
    logger.info(f"this is message # {i}")
    time.sleep(1)
```

Here in this short video you can see that the logs are being printed every second and its also being stored as log files.

```
ex5.py > ...
4
5 logger = logging.getLogger(__name__)
6 logger.setLevel(logging.INFO)
7
8 formatter = logging.Formatter(
9     "%(asctime)s: %(levelname)s: %(name)s: %(module)s: %(message)s"
10 )
11 timed_handler = TimedRotatingFileHandler(

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
~/engineering/logging_example
> python3 ex5.py
2022-01-10 01:39:08,211: INFO: __main__: ex5: this is message # 0
2022-01-10 01:39:09,216: INFO: __main__: ex5: this is message # 1
```

State of the directory after log files are generated.

```
~/engineering/logging_example
> tree
.
├── ex1.py
├── ex2.py
├── ex3.py
├── ex4.py
├── ex5.log
├── ex5.log.2022-01-10_01-38-54
├── ex5.log.2022-01-10_01-38-55
├── ex5.log.2022-01-10_01-38-56
├── ex5.log.2022-01-10_01-38-57
├── ex5.log.2022-01-10_01-38-58
├── ex5.log.2022-01-10_01-39-08
├── ex5.log.2022-01-10_01-39-09
├── ex5.log.2022-01-10_01-39-10
├── ex5.log.2022-01-10_01-39-11
├── ex5.log.2022-01-10_01-39-12
└── ex5.py

0 directories, 16 files
```