Python Logging Module

DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 040 – 64 51 27 86, 80 96 96 96 96, 92 46 21 21 43 | www.durgasoft.com

Python Logging

It is highly recommended to store complete application flow and exceptions information to a file. This process is called logging.

The main advanatages of logging are:

1. We can use log files while performing debugging
2. We can provide statistics like number of requests per day etc

To implement logging, Python provides inbuilt module logging.

Logging Levels:

Depending on type of information, logging data is divided according to the following 6 levels in python

1. CRITICAL===>50
Represents a very serious problem that needs high attention

2. ERROR ===>40
Represents a serious error

3. WARNING ==>30
Represents a warning message, some caution needed. It is alert to the programmer.

4. INFO==>20
Represents a message with some important information

5. DEBUG ===>10
Represents a message with debugging information

6. NOTSET==>0
Represents that level is not set

By default while executing Python program only WARNING and higher level messages will be displayed.

How to implement Logging:

To perform logging, first we required to create a file to store messages and we have to specify which level messages required to store.

We can do this by using basicConfig() function of logging module.

```
logging.basicConfig(filename='log.txt',level=logging.WARNING)
```

The above line will create a file log.txt and we can store either WARNING level or higher level messages to that file.

After creating log file, we can write messages to that file by using the following methods

```
logging.debug(message)
logging.info(message)
logging.warning(message)
logging.error(message)
logging.critical(message)
```

Q. Write a Python Program to create a log file and write WARNING and Higher level messages?

```
1) import logging
2) logging.basicConfig(filename='log.txt',level=logging.WARNING)
3) print('Logging Demo')
4) logging.debug('Debug Information')
5) logging.info('info Information')
6) logging.warning('warning Information')
7) logging.error('error Information')
8) logging.critical('critical Information')
```

log.txt:

```
WARNING:root:warning Information
ERROR:root:error Information
CRITICAL:root:critical Information
```

Note:

In the above program only WARNING and higher level messages will be written to the log file. If we set level as DEBUG then all messages will be written to the log file.

test.py:

```
1) import logging
2) logging.basicConfig(filename='log.txt',level=logging.DEBUG)
3) print('Logging Demo')
4) logging.debug('Debug Information')
5) logging.info('info Information')
6) logging.warning('warning Information')
```

```
7)  logging.error('error Information')
8)  logging.critical('critical Information')
```

log.txt:

```
DEBUG:root:Debug Information
INFO:root:info Information
WARNING:root:warning Information
ERROR:root:error Information
CRITICAL:root:critical Information
```

How to configure log file in over writing mode:

In the above program by default data will be appended to the log file.i.e append is the default mode. Instead of appending if we want to over write data then we have to use filemode property.

```
logging.basicConfig(filename='log786.txt',level=logging.WARNING)
```
meant for appending

```
logging.basicConfig(filename='log786.txt',level=logging.WARNING,filemode='a')
```
explicitly we are specifying appending.

```
logging.basicConfig(filename='log786.txt',level=logging.WARNING,filemode='w')
```
meant for over writing of previous data.

Note:

```
logging.basicConfig(filename='log.txt',level=logging.DEBUG)
```

If we are not specifying level then the default level is WARNING(30)

If we are not specifying file name then the messages will be printed to the console.

test.py:

```
1)  import logging
2)  logging.basicConfig()
3)  print('Logging Demo')
4)  logging.debug('Debug Information')
5)  logging.info('info Information')
6)  logging.warning('warning Information')
7)  logging.error('error Information')
8)  logging.critical('critical Information')
```

```
D:\durgaclasses>py test.py
Logging Demo
WARNING:root:warning Information
ERROR:root:error Information
CRITICAL:root:critical Information
```

How to Format log messages:

By using format keyword argument, we can format messages.

1. To display only level name:

```
logging.basicConfig(format='%(levelname)s')
```

Output:
WARNING
ERROR
CRITICAL

2. To display levelname and message:

```
logging.basicConfig(format='%(levelname)s:%(message)s')
```

Output:
WARNING:warning Information
ERROR:error Information
CRITICAL:critical Information

How to add timestamp in the log messages:

```
logging.basicConfig(format='%(asctime)s:%(levelname)s:%(message)s')
```

Output:
2018-06-15 11:50:08,325:WARNING:warning Information
2018-06-15 11:50:08,372:ERROR:error Information
2018-06-15 11:50:08,372:CRITICAL:critical Information

How to change date and time format:

We have to use special keyword argument: datefmt

```
logging.basicConfig(format='%(asctime)s:%(levelname)s:%(message)s', datefmt='%d/%m/%Y
%I:%M:%S %p')
```

datefmt='%d/%m/%Y %I:%M:%S %p' ===>case is important

Output:
15/06/2018 12:04:31 PM:WARNING:warning Information
15/06/2018 12:04:31 PM:ERROR:error Information
15/06/2018 12:04:31 PM:CRITICAL:critical Information

Note:
%I--->means 12 Hours time scale
%H--->means 24 Hours time scale

Eg:
logging.basicConfig(format='%(asctime)s:%(levelname)s:%(message)s', datefmt='%d/%m/%Y %H:%M:%S')

Output:
15/06/2018 12:06:28:WARNING:warning Information
15/06/2018 12:06:28:ERROR:error Information
15/06/2018 12:06:28:CRITICAL:critical Information

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

https://docs.python.org/3/library/time.html#time.strftime

How to write Python program exceptions to the log file:

By using the following function we can write exception information to the log file.

logging.exception(msg)

Q. Python Program to write exception information to the log file:

```
1)  import logging
2)  logging.basicConfig(filename='mylog.txt',level=logging.INFO,format='%(asctime)s:%(levelname)s:%(message)s',datefmt='%d/%m/%Y  %I:%M:%S %p')
3)  logging.info('A new Request Came')
4)  try:
5)      x=int(input('Enter First Number:'))
6)      y=int(input('Enter Second Number:'))
7)      print('The Result:',x/y)
8)
9)  except ZeroDivisionError as msg:
10)     print('cannot divide with zero')
11)     logging.exception(msg)
12)
13) except ValueError as msg:
14)     print('Please provide int values only')
15)     logging.exception(msg)
16)
17) logging.info('Request Processing Completed')
```

D:\durgaclasses>py test.py
Enter First Number:10
Enter Second Number:2
The Result: 5.0

DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 040 – 64 51 27 86, 80 96 96 96 96, 92 46 21 21 43 | www.durgasoft.com

```
D:\durgaclasses>py test.py
Enter First Number:20
Enter Second Number:2
The Result: 10.0

D:\durgaclasses>py test.py
Enter First Number:10
Enter Second Number:0
cannot divide with zero

D:\durgaclasses>py test.py
Enter First Number:ten
Please provide int values only
```

mylog.txt:

```
15/06/2018  12:30:51 PM:INFO:A new Request Came
15/06/2018  12:30:53 PM:INFO:Request Processing Completed
15/06/2018  12:30:55 PM:INFO:A new Request Came
15/06/2018  12:31:00 PM:INFO:Request Processing Completed
15/06/2018  12:31:02 PM:INFO:A new Request Came
15/06/2018  12:31:05 PM:ERROR:division by zero
Traceback (most recent call last):
  File "test.py", line 7, in <module>
    print('The Result:',x/y)
ZeroDivisionError: division by zero
15/06/2018  12:31:05 PM:INFO:Request Processing Completed
15/06/2018  12:31:06 PM:INFO:A new Request Came
15/06/2018  12:31:10 PM:ERROR:invalid literal for int() with base 10: 'ten'
Traceback (most recent call last):
  File "test.py", line 5, in <module>
    x=int(input('Enter First Number:'))
ValueError: invalid literal for int() with base 10: 'ten'
15/06/2018  12:31:10 PM:INFO:Request Processing Completed
```

Problems with root logger:

If we are not defining our own logger,then bydefault root logger will be considered.
Once we perform basic configuration to root logger then the configurations are fixed and we cannot change.

Demo Application:

student.py:

```
1) import logging
2) logging.basicConfig(filename='student.log',level=logging.INFO)
3) logging.info('info message from student module')
```

test.py:

```
1) import logging
2) import student
3) logging.basicConfig(filename='test.log',level=logging.DEBUG)
4) logging.debug('debug message from test module')
```

student.log:

INFO:root:info message from student module

In the above application the configurations performed in test module won't be reflected,b'z root logger is already configured in student module.

Need of Our own customized logger:

The problems with root logger are:

1. Once we set basic configuration then that configuration is final and we cannot change

2. It will always work for only one handler at a time, either console or file, but not both simultaneously

3. It is not possible to configure logger with different configurations at different levels

4. We cannot specify multiple log files for multiple modules/classes/methods.

To overcome these problems we should go for our own customized loggers

Advanced logging Module Features: Logger:

Logger is more advanced than basic logging.
It is highly recommended to use and it provides several extra features.

Steps for Advanced Logging:

1. Creation of Logger object and set log level

```
logger = logging.getLogger('demologger')
logger.setLevel(logging.INFO)
```

2. Creation of Handler object and set log level
There are several types of Handlers like StreamHandler, FileHandler etc

```
consoleHandler = logging.StreamHandler()
consoleHandler.setLevel(logging.INFO)
```

Note: If we use StreamHandler then log messages will be printed to console

3. Creation of Formatter object

```
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s: %(message)s',
datefmt='%d/%m/%Y %I:%M:%S %p')
```

4. Add Formatter to Handler

```
consoleHandler.setFormatter(formatter)
```

5. Add Handler to Logger

```
logger.addHandler(consoleHandler)
```

6. Write messages by using logger object and the following methods

```
logger.debug('debug message')
logger.info('info message')
logger.warn('warn message')
logger.error('error message')
logger.critical('critical message')
```

Note: Bydefault logger will set to WARNING level. But we can set our own level based on our requirement.

```
logger = logging.getLogger('demologger')
logger.setLevel(logging.INFO)
```

logger log level by default available to console and file handlers. If we are not satisfied with logger level, then we can set log level explicitly at console level and file levels.

```
consoleHandler = logging.StreamHandler()
consoleHandler.setLevel(logging.WARNING)
```

```
fileHandler=logging.FileHandler('abc.log',mode='a')
fileHandler.setLevel(logging.ERROR)
```

Note:

console and file log levels should be supported by logger. i.e logger log level should be lower than console and file levels. Otherwise only logger log level will be considered.

Eg:

logger==>DEBUG console===>INFO -------->Valid and INFO will be considered
logger==>INFO console===>DEBUG -------->Invalid and only INFO will be considered to the console.

Demo Program for Console Handler:

```
1)  import logging
2)  class LoggerDemoConsole:
3)
4)      def testLog(self):
5)          logger = logging.getLogger('demologger')
6)          logger.setLevel(logging.INFO)
7)
8)          consoleHandler = logging.StreamHandler()
9)          consoleHandler.setLevel(logging.INFO)
10)
11)         formatter = logging.Formatter('%(asctime)s - %(name)s -
        %(levelname)s: %(message)s',
12)                 datefmt='%m/%d/%Y %I:%M:%S %p')
13)
14)         consoleHandler.setFormatter(formatter)
15)         logger.addHandler(consoleHandler)
16)
17)         logger.debug('debug message')
18)         logger.info('info message')
19)         logger.warn('warn message')
20)         logger.error('error message')
21)         logger.critical('critical message')
22)
23) demo = LoggerDemoConsole()
24) demo.testLog()
```

```
D:\durgaclasses>py loggingdemo3.py
06/18/2018 12:14:15 PM - demologger - INFO: info message
06/18/2018 12:14:15 PM - demologger - WARNING: warn message
06/18/2018 12:14:15 PM - demologger - ERROR: error message
06/18/2018 12:14:15 PM - demologger - CRITICAL: critical message
```

Note:

If we want to use class name as logger name then we have to create logger object as follows

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

In this case output is:

```
D:\durgaclasses>py loggingdemo3.py
06/18/2018 12:21:00 PM - LoggerDemoConsole - INFO: info message
06/18/2018 12:21:00 PM - LoggerDemoConsole - WARNING: warn message
06/18/2018 12:21:00 PM - LoggerDemoConsole - ERROR: error message
06/18/2018 12:21:00 PM - LoggerDemoConsole - CRITICAL: critical message
```

Demo Program for File Handler:

```
1)  import logging
2)  class LoggerDemoConsole:
3)
4)      def testLog(self):
5)          logger = logging.getLogger('demologger')
6)          logger.setLevel(logging.INFO)
7)
8)          fileHandler = logging.FileHandler('abc.log',mode='a')
9)          fileHandler.setLevel(logging.INFO)
10)
11)         formatter = logging.Formatter('%(asctime)s - %(name)s -
        %(levelname)s: %(message)s',
12)                 datefmt='%m/%d/%Y %I:%M:%S %p')
13)
14)         fileHandler.setFormatter(formatter)
15)         logger.addHandler(fileHandler)
16)
17)         logger.debug('debug message')
18)         logger.info('info message')
19)         logger.warn('warn message')
20)         logger.error('error message')
21)         logger.critical('critical message')
22)
23) demo = LoggerDemoConsole()
24) demo.testLog()
```

abc.log:

```
07/05/2018 08:58:04 AM - demologger - INFO: info message
07/05/2018 08:58:04 AM - demologger - WARNING: warn message
07/05/2018 08:58:04 AM - demologger - ERROR: error message
07/05/2018 08:58:04 AM - demologger - CRITICAL: critical message
```

Logger with Configuration File:

In the above program, everything we hard coded in the python script. It is not a good programming practice. We will configure all the required things inside a configuration file and we can use this file directly in our program.

```
logging.config.fileConfig('logging.conf')
logger = logging.getLogger(LoggerDemoConf.__name__)
```

Note: The extension of the file need not be conf. We can use any extension like txt or durga etc

logging.conf:

```
[loggers]
keys=root,LoggerDemoConf

[handlers]
keys=fileHandler

[formatters]
keys=simpleFormatter

[logger_root]
level=DEBUG
handlers=fileHandler

[logger_LoggerDemoConf]
level=DEBUG
handlers=fileHandler
qualname=demoLogger

[handler_fileHandler]
class=FileHandler
level=DEBUG
formatter=simpleFormatter
args=('test.log', 'w')

[formatter_simpleFormatter]
format=%(asctime)s - %(name)s - %(levelname)s - %(message)s
datefmt=%m/%d/%Y %I:%M:%S %p
```

test.py:

```
1)  import logging
2)  import logging.config
3)  class LoggerDemoConf():
4)
5)      def testLog(self):
6)          logging.config.fileConfig('logging.conf')
7)          logger = logging.getLogger(LoggerDemoConf.__name__)
8)
9)          logger.debug('debug message')
10)         logger.info('info message')
11)         logger.warn('warn message')
12)         logger.error('error message')
13)         logger.critical('critical message')
14)
15) demo = LoggerDemoConf()
16) demo.testLog()
```

test.log:

```
06/18/2018 12:40:05 PM - LoggerDemoConf - DEBUG - debug message
06/18/2018 12:40:05 PM - LoggerDemoConf - INFO - info message
06/18/2018 12:40:05 PM - LoggerDemoConf - WARNING - warn message
06/18/2018 12:40:05 PM - LoggerDemoConf - ERROR - error message
06/18/2018 12:40:05 PM - LoggerDemoConf - CRITICAL - critical message
```

Case-1: To set log level as INFO:

```
[handler_fileHandler]
class=FileHandler
level=INFO
formatter=simpleFormatter
args=('test.log', 'w')
```

Case-2: To set Append Mode:

```
[handler_fileHandler]
class=FileHandler
level=INFO
formatter=simpleFormatter
args=('test.log', 'a')
```

Creation of Custom Logger:

customlogger.py:

```
1)  import logging
2)  import inspect
3)  def getCustomLogger(level):
4)          # Get Name of class/method from where this method called
5)      loggername=inspect.stack()[1][3]
6)      logger=logging.getLogger(loggername)
7)      logger.setLevel(level)
8)
9)      fileHandler=logging.FileHandler('abc.log',mode='a')
10)     fileHandler.setLevel(level)
11)
12)     formatter = logging.Formatter('%(asctime)s - %(name)s -
        %(levelname)s: %(message)s',datefmt='%m/%d/%Y %I:%M:%S %p')
13)     fileHandler.setFormatter(formatter)
14)     logger.addHandler(fileHandler)
15)
16)     return logger
```

test.py:

```
1)  import logging
2)  from customlogger import getCustomLogger
3)  class LoggingDemo:
4)     def m1(self):
5)        logger=getCustomLogger(logging.DEBUG)
6)        logger.debug('m1:debug message')
7)        logger.info('m1:info message')
8)        logger.warn('m1:warn message')
9)        logger.error('m1:error message')
10)       logger.critical('m1:critical message')
11)    def m2(self):
12)       logger=getCustomLogger(logging.WARNING)
13)       logger.debug('m2:debug message')
14)       logger.info('m2:info message')
15)       logger.warn('m2:warn message')
16)       logger.error('m2:error message')
17)       logger.critical('m2:critical message')
18)    def m3(self):
19)       logger=getCustomLogger(logging.ERROR)
20)       logger.debug('m3:debug message')
21)       logger.info('m3:info message')
22)       logger.warn('m3:warn message')
23)       logger.error('m3:error message')
24)       logger.critical('m3:critical message')
25)
26) l=LoggingDemo()
27) print('Custom Logger Demo')
28) l.m1()
29) l.m2()
30) l.m3()
```

abc.log:

```
06/19/2018 12:17:19 PM - m1 - DEBUG: m1:debug message
06/19/2018 12:17:19 PM - m1 - INFO: m1:info message
06/19/2018 12:17:19 PM - m1 - WARNING: m1:warn message
06/19/2018 12:17:19 PM - m1 - ERROR: m1:error message
06/19/2018 12:17:19 PM - m1 - CRITICAL: m1:critical message
06/19/2018 12:17:19 PM - m2 - WARNING: m2:warn message
06/19/2018 12:17:19 PM - m2 - ERROR: m2:error message
06/19/2018 12:17:19 PM - m2 - CRITICAL: m2:critical message
06/19/2018 12:17:19 PM - m3 - ERROR: m3:error message
06/19/2018 12:17:19 PM - m3 - CRITICAL: m3:critical message
```

How to create seperate log file Based on Caller:

```
1)  import logging
2)  import inspect
3)  def getCustomLogger(level):
4)      loggername=inspect.stack()[1][3]
5)      logger=logging.getLogger(loggername)
6)      logger.setLevel(level)
7)
8)      fileHandler=logging.FileHandler('{}.log'.format(loggername),mode='a')
9)      fileHandler.setLevel(level)
10)
11)     formatter = logging.Formatter('%(asctime)s - %(name)s -
        %(levelname)s: %(message)s',datefmt='%m/%d/%Y %I:%M:%S %p')
12)     fileHandler.setFormatter(formatter)
13)     logger.addHandler(fileHandler)
14)
15)     return logger
```

test.py:

#Same as previous

```
1)  import logging
2)  from customlogger import getCustomLogger
3)  class LoggingDemo:
4)      def m1(self):
5)          logger=getCustomLogger(logging.DEBUG)
6)          logger.debug('m1:debug message')
7)          logger.info('m1:info message')
8)          logger.warn('m1:warn message')
9)          logger.error('m1:error message')
10)         logger.critical('m1:critical message')
11)     def m2(self):
12)         logger=getCustomLogger(logging.WARNING)
13)         logger.debug('m2:debug message')
14)         logger.info('m2:info message')
15)         logger.warn('m2:warn message')
16)         logger.error('m2:error message')
17)         logger.critical('m2:critical message')
18)     def m3(self):
19)         logger=getCustomLogger(logging.ERROR)
20)         logger.debug('m3:debug message')
21)         logger.info('m3:info message')
22)         logger.warn('m3:warn message')
23)         logger.error('m3:error message')
24)         logger.critical('m3:critical message')
25)
```

```
26) l=LoggingDemo()
27) print('Logging Demo with Seperate Log File')
28) l.m1()
29) l.m2()
30) l.m3()
```

m1.log:

```
06/19/2018 12:26:04 PM - m1 - DEBUG: m1:debug message
06/19/2018 12:26:04 PM - m1 - INFO: m1:info message
06/19/2018 12:26:04 PM - m1 - WARNING: m1:warn message
06/19/2018 12:26:04 PM - m1 - ERROR: m1:error message
06/19/2018 12:26:04 PM - m1 - CRITICAL: m1:critical message
```

m2.log:

```
06/19/2018 12:26:04 PM - m2 - WARNING: m2:warn message
06/19/2018 12:26:04 PM - m2 - ERROR: m2:error message
06/19/2018 12:26:04 PM - m2 - CRITICAL: m2:critical message
```

m3.log:

```
06/19/2018 12:26:04 PM - m3 - ERROR: m3:error message
06/19/2018 12:26:04 PM - m3 - CRITICAL: m3:critical message
```

Advantages of customized logger:

1. We can reuse same customlogger code where ever logger required.
2. For every caller we can able to create a seperate log file
3. For different handlers we can set different log levels.

Another Example for Custom Handler:

customlogger.py:

```
1)  import logging
2)  import inspect
3)  def getCustomLogger(level):
4)      loggername=inspect.stack()[1][3]
5)
6)      logger=logging.getLogger(loggername)
7)      logger.setLevel(level)
8)      fileHandler=logging.FileHandler('test.log',mode='a')
9)      fileHandler.setLevel(level)
10)     formatter=logging.Formatter('%(asctime)s - %(name)s -
        %(levelname)s: %(message)s',datefmt='%m/%d/%Y %I:%M:%S %p')
11)     fileHandler.setFormatter(formatter)
12)     logger.addHandler(fileHandler)
13)     return logger
```

test.py:

```
1)  import logging
2)  from customlogger import getCustomLogger
3)  class Test:
4)      def logtest(self):
5)          logger=getCustomLogger(logging.DEBUG)
6)          logger.debug('debug  message')
7)          logger.info('info  message')
8)          logger.warning('warning  message')
9)          logger.error('error  message')
10)         logger.critical('critical message')
11) t=Test()
12) t.logtest()
```

student.py:

```
1)  import logging
2)  from customlogger import getCustomLogger
3)  def studentfunction():
4)      logger=getCustomLogger(logging.ERROR)
5)      logger.debug('debug  message')
6)      logger.info('info  message')
7)      logger.warning('warning  message')
8)      logger.error('error  message')
9)      logger.critical('critical message')
10) studentfunction()
```

Note: we can disable a partcular level of logging as follows:
logging.disable(logging.CRITICAL)