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

test.py:

```
1) import logging
2) logger=logging.getLogger('demologger')
3) logger.setLevel(logging.DEBUG)
4)
5) consoleHandler = logging.StreamHandler()
6)
7) formatter=logging.Formatter('%(asctime)s:%(levelname)s:%(name)s:%(message)s',
   datefmt='%d/%m/%Y  %I:%M:%S %p')
8)
9) consoleHandler.setFormatter(formatter)
10) logger.addHandler(consoleHandler)
11)
12) logger.critical('It is critical message')
13) logger.error('It is error message')
14) logger.warning('It is warning message')
15) logger.info('It is info message')
16) logger.debug('It is debug message')
```

output

```
D:\durgaclasses>py test.py
27/07/2019  11:13:20 PM:CRITICAL:demologger:It is critical message
27/07/2019  11:13:20 PM:ERROR:demologger:It is error message
27/07/2019  11:13:20 PM:WARNING:demologger:It is warning message
27/07/2019  11:13:20 PM:INFO:demologger:It is info message
27/07/2019  11:13:20 PM:DEBUG:demologger:It is debug message
```

Demo Program for File Handler:

test.py

```
1) import logging
2) logger=logging.getLogger('demologger')
3) logger.setLevel(logging.DEBUG)
4)
5) fileHandler=logging.FileHandler('custtest.log',mode='w')
6)
7) formatter=logging.Formatter('%(asctime)s:%(levelname)s:%(name)s:%(message)s',
   datefmt='%d/%m/%Y  %I:%M:%S %p')
8)
9) fileHandler.setFormatter(formatter)
```

```
10) logger.addHandler(fileHandler)
11)
12) logger.critical('It is critical message')
13) logger.error('It is error message')
14) logger.warning('It is warning message')
15) logger.info('It is info message')
16) logger.debug('It is debug message')
```

custtest.log

```
27/07/2019  11:16:39 PM:CRITICAL:demologger:It is critical message
27/07/2019  11:16:39 PM:ERROR:demologger:It is error message
27/07/2019  11:16:39 PM:WARNING:demologger:It is warning message
27/07/2019  11:16:39 PM:INFO:demologger:It is info message
27/07/2019  11:16:39 PM:DEBUG:demologger:It is debug message
```

Demo Program to use both Console and File Handlers:

test.py

```
1)  import logging
2)  logger=logging.getLogger('demo logger')
3)  logger.setLevel(logging.INFO)
4)
5)  consoleHandler = logging.StreamHandler()
6)  fileHandler=logging.FileHandler('abc.log',mode='w')
7)
8)  formatter=logging.Formatter('%(asctime)s:%(levelname)s:%(name)s:%(message)s',
    datefmt='%d/%m/%Y  %I:%M:%S %p')
9)
10) consoleHandler.setFormatter(formatter)
11) fileHandler.setFormatter(formatter)
12)
13) logger.addHandler(consoleHandler)
14) logger.addHandler(fileHandler)
15)
16) logger.critical('It is critical message')
17) logger.error('It is error message')
18) logger.warning('It is warning message')
19) logger.info('It is info message')
20) logger.debug('It is debug message')
```

output on console

```
D:\durgaclasses>py test.py
27/07/2019  11:19:41 PM:CRITICAL:demo logger:It is critical message
```

27/07/2019 11:19:41 PM:ERROR:demo logger:It is error message
27/07/2019 11:19:41 PM:WARNING:demo logger:It is warning message
27/07/2019 11:19:41 PM:INFO:demo logger:It is info message

abc.log

27/07/2019 11:19:41 PM:CRITICAL:demo logger:It is critical message
27/07/2019 11:19:41 PM:ERROR:demo logger:It is error message
27/07/2019 11:19:41 PM:WARNING:demo logger:It is warning message
27/07/2019 11:19:41 PM:INFO:demo logger:It is info message

Demo program to define and use custom logger with different modules and with different log files:

test.py

```
1) import logging
2) import student
3) logger=logging.getLogger('testlogger')
4) logger.setLevel(logging.DEBUG)
5)
6) fileHandler=logging.FileHandler('test.log',mode='a')
7)
8) formatter=logging.Formatter('%(asctime)s:%(levelname)s:%(name)s:%(message)s',
   datefmt='%d/%m/%Y  %I:%M:%S %p')
9)
10) fileHandler.setFormatter(formatter)
11) logger.addHandler(fileHandler)
12)
13) logger.critical('critical message from test module')
14) logger.error('error message from test module')
15) logger.warning('warning message from test module')
16) logger.info('info message from test module')
17) logger.debug('debug message from test module')
```

student.py

```
1) import logging
2) logger=logging.getLogger('studentlogger')
3) logger.setLevel(logging.DEBUG)
4)
5) fileHandler=logging.FileHandler('student.log',mode='a')
6) fileHandler.setLevel(logging.ERROR)
7)
```

```
8) formatter=logging.Formatter('%(asctime)s:%(levelname)s:%(name)s:%(message)s',
   datefmt='%d/%m/%Y  %H:%M:%S')
9)
10) fileHandler.setFormatter(formatter)
11) logger.addHandler(fileHandler)
12)
13) logger.critical('critical message from student module')
14) logger.error('error message from student module')
15) logger.warning('warning message student test module')
16) logger.info('info message from student module')
17) logger.debug('debug message from student module')
```

test.log

```
24/07/2019 01:24:09 PM:demologger:CRITICAL:It is critical message
24/07/2019 01:24:09 PM:demologger:ERROR:It is error message
27/07/2019  11:27:06 PM:CRITICAL:testlogger:critical message from test module
27/07/2019  11:27:06 PM:ERROR:testlogger:error message from test module
27/07/2019  11:27:06 PM:WARNING:testlogger:warning message from test module
27/07/2019  11:27:06 PM:INFO:testlogger:info message from test module
27/07/2019  11:27:06 PM:DEBUG:testlogger:debug message from test module
```

student.log

```
2019-07-22 13:52:21,343:CRITICAL:studentlogger:critical message from student module
2019-07-22 13:52:21,343:ERROR:studentlogger:error message from student module
27/07/2019  23:26:35:CRITICAL:studentlogger:critical message from student module
27/07/2019  23:26:35:ERROR:studentlogger:error message from student module
27/07/2019  23:26:59:CRITICAL:studentlogger:critical message from student module
27/07/2019  23:26:59:ERROR:studentlogger:error message from student module
27/07/2019  23:27:06:CRITICAL:studentlogger:critical message from student module
27/07/2019  23:27:06:ERROR:studentlogger:error message from student module
```

Note: In the above program we are maintaining different log files for different modules, which is not possible by root logger.

Creation of generic custom logger and usage Demo Program-1:

custlogger.py

```
1) import logging
2) import inspect
3) def get_custom_logger(level):
4)     function_name =inspect.stack()[1][3]
5)     logger_name=function_name+" logger"
6)
```

```
7)    logger=logging.getLogger(logger_name)
8)    logger.setLevel(level)
9)
10)   fileHandler = logging.FileHandler('abc.log',mode='a')
11)   fileHandler.setLevel(level)
12)   formatter=logging.Formatter(
13)       '%(asctime)s:%(levelname)s:%(name)s:%(message)s',
14)        datefmt='%d/%m/%Y  %I:%M:%S %p')
15)   fileHandler.setFormatter(formatter)
16)   logger.addHandler(fileHandler)
17)   return logger
```

test.py

```
1)  from custlogger import get_custom_logger
2)  import logging
3)  def logtest():
4)     logger=get_custom_logger(logging.DEBUG)
5)     logger.critical('critical message from test module')
6)     logger.error('error message from test module')
7)     logger.warning('warning message from test module')
8)     logger.info('info message from test module')
9)     logger.debug('debug message from test module')
10) logtest()
```

student.py

```
1)  from custlogger import get_custom_logger
2)  import logging
3)  def logstudent():
4)     logger=get_custom_logger(logging.ERROR)
5)     logger.critical('critical message from student module')
6)     logger.error('error message from student module')
7)     logger.warning('warning message from student module')
8)     logger.info('info message from student module')
9)     logger.debug('debug message from student module')
10) logstudent()
```

abc.log

```
27/07/2019  11:35:53 PM:CRITICAL:logtest logger:critical message from test module
27/07/2019  11:35:53 PM:ERROR:logtest logger:error message from test module
27/07/2019  11:35:53 PM:WARNING:logtest logger:warning message from test module
27/07/2019  11:35:53 PM:INFO:logtest logger:info message from test module
27/07/2019  11:35:53 PM:DEBUG:logtest logger:debug message from test module
```

27/07/2019 11:35:54 PM:CRITICAL:logstudent logger:critical message from student module
27/07/2019 11:35:54 PM:ERROR:logstudent logger:error message from student module

Creation of generic custom logger and usage Demo Program-2:

custlogger.py

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

test.py

```
1)  from custlogger import get_custom_logger
2)  import logging
3)  def f1():
4)      logger=get_custom_logger(logging.DEBUG)
5)      logger.critical('critical message from f1')
6)      logger.error('error message from f1')
7)      logger.warning('warning message from f1')
8)      logger.info('info message from f1')
9)      logger.debug('debug message from f1')
10) def f2():
11)     logger=get_custom_logger(logging.WARNING)
12)     logger.critical('critical message from f2')
13)     logger.error('error message from f2')
14)     logger.warning('warning message from f2')
15)     logger.info('info message from f2')
16)     logger.debug('debug message from f2')
```

```
17) def f3():
18)     logger=get_custom_logger(logging.ERROR)
19)     logger.critical('critical message from f3')
20)     logger.error('error message from f3')
21)     logger.warning('warning message from f3')
22)     logger.info('info message from f3')
23)     logger.debug('debug message from f3')
24) f1()
25) f2()
26) f3()
```

abc.log

```
27/07/2019  11:38:56 PM:CRITICAL:f1 logger:critical message from f1
27/07/2019  11:38:56 PM:ERROR:f1 logger:error message from f1
27/07/2019  11:38:56 PM:WARNING:f1 logger:warning message from f1
27/07/2019  11:38:56 PM:INFO:f1 logger:info message from f1
27/07/2019  11:38:56 PM:DEBUG:f1 logger:debug message from f1
27/07/2019  11:38:56 PM:CRITICAL:f2 logger:critical message from f2
27/07/2019  11:38:56 PM:ERROR:f2 logger:error message from f2
27/07/2019  11:38:56 PM:WARNING:f2 logger:warning message from f2
27/07/2019  11:38:56 PM:CRITICAL:f3 logger:critical message from f3
27/07/2019  11:38:56 PM:ERROR:f3 logger:error message from f3
```

How to create separate log file based on caller dynamically?

custlogger.py

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

test.py

```
1)  from custlogger import get_custom_logger
2)  import logging
3)  def f1():
4)      logger=get_custom_logger(logging.DEBUG)
5)      logger.critical('critical message from f1')
6)      logger.error('error message from f1')
7)      logger.warning('warning message from f1')
8)      logger.info('info message from f1')
9)      logger.debug('debug message from f1')
10) def f2():
11)     logger=get_custom_logger(logging.WARNING)
12)     logger.critical('critical message from f2')
13)     logger.error('error message from f2')
14)     logger.warning('warning message from f2')
15)     logger.info('info message from f2')
16)     logger.debug('debug message from f2')
17) def f3():
18)     logger=get_custom_logger(logging.ERROR)
19)     logger.critical('critical message from f3')
20)     logger.error('error message from f3')
21)     logger.warning('warning message from f3')
22)     logger.info('info message from f3')
23)     logger.debug('debug message from f3')
24) f1()
25) f2()
26) f3()
```

f1.log

```
27/07/2019  11:41:33 PM:CRITICAL:f1 logger:critical message from f1
27/07/2019  11:41:33 PM:ERROR:f1 logger:error message from f1
27/07/2019  11:41:33 PM:WARNING:f1 logger:warning message from f1
27/07/2019  11:41:33 PM:INFO:f1 logger:info message from f1
27/07/2019  11:41:33 PM:DEBUG:f1 logger:debug message from f1
```

f2.log

```
27/07/2019  11:41:33 PM:CRITICAL:f2 logger:critical message from f2
27/07/2019  11:41:33 PM:ERROR:f2 logger:error message from f2
27/07/2019  11:41:33 PM:WARNING:f2 logger:warning message from f2
```

f3.log

```
27/07/2019  11:41:33 PM:CRITICAL:f3 logger:critical message from f3
27/07/2019  11:41:33 PM:ERROR:f3 logger:error message from f3
```

Need of separating logger configurations into a file or dict or JSON or YAML?

Instead of hard coding logging configurations inside our application, we can separate into into a file or dict or JSON or YAML.

Advantages:

1. Modifications will become very easy.
2. We can reuse same configurations in different modules.
3. Length of the code will be reduced and readability will be improved.

Demo Program for Logger configurations into a separate config file

logging_config.init: For File Handler

```
1)  [loggers]
2)  keys=root,demologger
3)
4)  [handlers]
5)  keys=fileHandler
6)
7)  [formatters]
8)  keys=sampleFormatter
9)
10) [logger_root]
11) level=DEBUG
12) handlers=fileHandler
13)
14) [logger_demologger]
15) level=DEBUG
16) handlers=fileHandler
17) qualname=demoLogger
18)
19)  [handler_fileHandler]
20) class=FileHandler
21) level=DEBUG
22) formatter=sampleFormatter
23) args=('test.log','w')
24)
25) [formatter_sampleFormatter]
26) format=%(asctime)s:%(name)s:%(levelname)s:%(message)s
27) datefmt=%d/%m/%Y %I:%M:%S %p
```

logging_config.init: For Console Handler

```
[loggers]
keys=root,demologger

[handlers]
keys=consoleHandler

[formatters]
keys=sampleFormatter

[logger_root]
level=DEBUG
handlers=consoleHandler

[logger_demologger]
level=DEBUG
handlers=consoleHandler
qualname=demoLogger


[handler_consoleHandler]
class=StreamHandler
level=DEBUG
formatter=sampleFormatter
args=(sys.stdout,)

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

test.py

```
import logging
import logging.config
logging.config.fileConfig("logging_config.init")
logger=logging.getLogger('demologger')
logger.critical('It is critical message')
logger.error('It is error message')
logger.warning('It is warning message')
logger.info('It is info message')
logger.debug('It is debug message')
```

Logger configurations into a dictionary

test.py

```
1)  import logging
2)  from logging.config import dictConfig
3)
4)  logging_config = dict(
5)     version = 1,
6)     formatters = {
7)        'f': {'format':
8)             '%(asctime)s:%(name)s:%(levelname)s:%(message)s',
9)             'datefmt':'%d/%m/%Y %I:%M:%S %p'}
10)       },
11)    handlers = {
12)       'h': {'class': 'logging.StreamHandler',
13)            'formatter': 'f',
14)            'level': logging.DEBUG}
15)       },
16)    root = {
17)       'handlers': ['h'],
18)       'level': logging.DEBUG,
19)       },
20) )
21)
22) dictConfig(logging_config)
23)
24) logger = logging.getLogger()
25) logger.critical('it is critical message')
26) logger.error('it is error message')
27) logger.warning('it is warning message')
28) logger.info('it is info message')
29) logger.debug('it is debug message')
```