Part 2: The Print Statement You Can Hide

Bookmark this page

The Print Statement You Can Hide

Let’s make a couple of changes to our code:

```
import logging

logging.basicConfig(level=logging.DEBUG)
def my_fun(n):
    for i in range(0, n):
        logging.debug(i)
        100 / (50 - i)

if __name__ == "__main__":
    my_fun(100)
```

We’ve imported the logging library, set some kind of logging configuration, and then replaced our print statement with a *logging.debug* statement.

Now running simple.py produces the following output:

```
$ python simple.py
DEBUG:root:0
DEBUG:root:1
DEBUG:root:2
...
DEBUG:root:48
DEBUG:root:49
DEBUG:root:50
Traceback (most recent call last):
  File "simple.py", line 10, in <module>
    my_fun(100)
  File "simple.py", line 7, in my_fun
    100 / (50 - i)
ZeroDivisionError: division by zero
```

So far, this doesn’t look very different from the print statement that we were using before. But let’s change one line of the script:

```
import logging

logging.basicConfig(level=logging.WARNING)  # Change the level to logging.WARNING
def my_fun(n):
    for i in range(0, n):
        logging.debug(i)
        100 / (50 - i)

if __name__ == "__main__":
    my_fun(100)
```

Now try running the script again:

```
$ python simple.py
Traceback (most recent call last):
  File "simple.py", line 10, in <module>
    my_fun(100)
  File "simple.py", line 7, in my_fun
    100 / (50 - i)
ZeroDivisionError: division by zero
```

What happened?

The logging library includes the idea of various *levels* of logging messages: some messages are more important than others. For example, if you were curious to know the values that a function was being called with, then you might put a logging statement into that function to help you understand when it was being called, and with what arguments. For example:

```
def my_fun(n):
    logging.info("Function my_fun called with value {}".format(n))
    do_something(n)
    ...
```

This logging statement is just giving us some information about how the function is being used, so we've used the *logging.info* method.

In our example script, when we were trying to figure out what value of *i* was causing our script to crash, we were debugging our code. That's why we used a *logging.debug* statement. Now that we know that the value 50 causes our code to crash, we could put in a *logging.warning* statement that will warn us of dangerous conditions:

```
import logging

logging.basicConfig(level=logging.WARNING)
def my_fun(n):
    for i in range(0, n):
        logging.debug(i)
        if i == 50:                                   # Add this line
            logging.warning("The value of i is 50.")  # Add this line
        100 / (50 - i)

if __name__ == "__main__":
    my_fun(100)
```

If we wanted to handle the division by zero error gracefully, then we could modify the code to attempt the *100 / (50 - i)* operation inside of a try, except block. Then we would log an *error* if our script did attempt to divide by 0:

```
import logging

logging.basicConfig(level=logging.WARNING)

def my_fun(n):
    for i in range(0, n):
        logging.debug(i)
        if i == 50:
            logging.warning("The value of i is 50.")
        try:
            100 / (50 - i)
        except ZeroDivisionError:
            logging.error("Tried to divide by zero. Var i was {}. Recovered
gracefully.".format(i))

if __name__ == "__main__":
    my_fun(100)
```

You can see all of the logging levels in the logging documentation. Each level has an associated logging method, like *logging.error*, *logging.warning*, etc.

Now what do we get when we run our code?

```
$ python simple.py
WARNING:root:The value of i is 50.
ERROR:root:Tried to divide by zero, i was 50. Recovered gracefully.
```

Why is it not showing the *logging.debug* statements?

Previous Next

The statement *logging.basicConfig(level=logging.WARNING) *tells the logger to *only* display log messages with level WARNING and above. Look back to the logging levels documentation. You'll see that the DEBUG level is below the WARNING level, so it won't be displayed. When we were debugging this code, the debug statements were helping us understand why our code was failing, but now it would be overwhelming to see them every time we run our code. We've *hidden* the statements by making a single configuration change.

The idea is that you might be working on a project with a lot of Python files. You may have put debugging or information statements into several of these files. While you're authoring the project, these messages are useful. And once you think you've worked out all of the bugs in your code, you don't have to go through all of your files and find every logging statement: you can just turn off the unimportant ones by setting the log level in your main script.

What is the default log level? If you don't specify a log level, then will you see *all* log messages, or is there some default level that the logging library will choose for you? To answer that, try running the following script:

```
# loggingtest.py
import logging

logging.critical("This is a critical error!")
logging.error("I'm an error.")
logging.warning("Hello! I'm a warning!")
logging.info("This is some information.")
```