Python
Special topic: Debugging and profiling

Prof. Dr. Thomas Kopinski

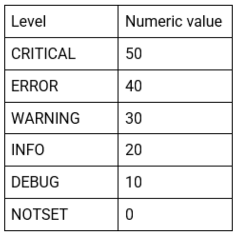
July 3, 2023

Abstract

This special exercise serves as an introduction into debugging and program/system profiling.

Task 1: Simple debugging techniques

As you might have noticed by now, a lot of the times you write a program/script it will not just work as expected from the beginning. Often some problems occur which we only notice once we actually test the written source code. There are different options to help you fix bugs in your code, the simplest one relies on just printing out the status of the program and certain variables at specific times during the execution of the code, preferably right before the crash. While this method works, at least to a certain degree, it can become quite cumbersome, especially once you deal with more complex programs and many print-statements designated to debugging as you probably want to remove them afterwards. This can become time consuming and tedious, so what is the next best option here?
Python includes a package called *logging*, which works similar to print() as it outputs a user-defined message at a user-defined point in the program, but additionally allows to use different, pre-defined levels of importance for each message (see fig. 1). This can become quite handy as it allows you to suppress the output of certain levels of messages depending on the environment you are running your program in (e.g. test, production etc.).



Level	Numeric value
CRITICAL	50
ERROR	40
WARNING	30
INFO	20
DEBUG	10
NOTSET	0

Figure 1: Logging levels offered by the *logging* package.

Take a look at "logging_example.py", which can be found on the GitHub repository, to see an example implementation of a logging object. It might seem like a lot of overhead when compared to a simple print() command, but it also shows you some more neat features and options you will not be able to pull off with print() on its own.

Exercises:

1.1

In the GitHub repo for this topic you can find a file named "simple_print.py". Rebuild its functionality by utilizing the *logging* package and log the output with a level of "WARNING" and above to a file called "simple_logging_log.log" as well as the sys.stdout.

Task 2: Debugging tools

Depending on the complexity of your program, you might want to use designated tools to handle the debugging process. Most IDE's come with some sort of debugging capabilities, but since we do not always have an IDE at hand we will focus on the python debugger (pdb) here which can be run from any command line.
Debuggers offer various further advantages when compared to print() or logging, some of them include:

- Halt execution at a given line
- Get value of variables, even after program crash
- Execute code line-by-line (one instruction at a time)

There is a neat, interactive tutorial for pdb on GitHub which will go into more details, including usage, commands and some theory which you have to work through in the exercises accompanying this task.

Exercises:

2.1

Clone, work through and understand this pdb tutorial (https://github.com/spiside/pdb-tutorial).

2.1

Use your newly gained knowledge to debug the dice game found in the aforementioned GitHub repository. Make sure that the game does not crash anymore as well as correctly counts the value of each dice afterwards.