

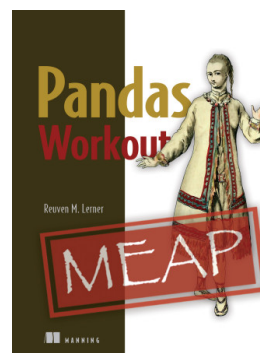
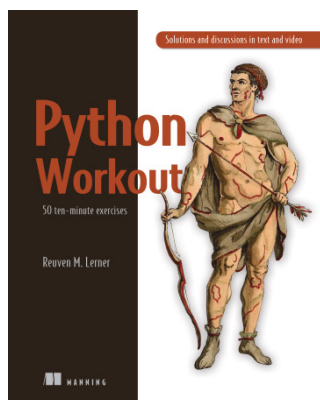
Stop using “print”! Using the “logging” module

Reuven M. Lerner • Euro Python 2023, Prague

<https://lerner.co.il> • reuven@lerner.co.il • @reuvenmlerner

I teach Python and Pandas

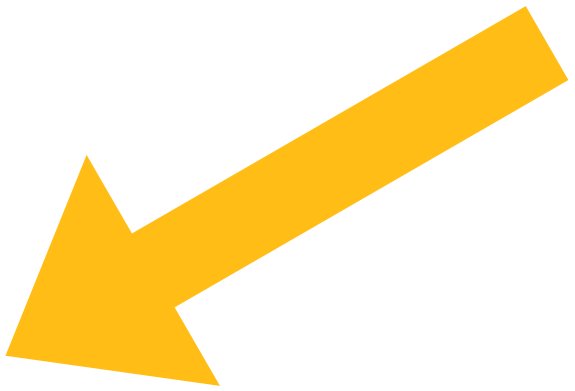
- For companies around the world
- Video courses including a boot camp
- Weekly Python Exercise
- Newsletters
 - Better Developers (Python articles)
 - Bamboo Weekly (Pandas puzzles)
- Books (Python Workout, Pandas Workout)



Have you ever written buggy code?

Let's write some!

```
def fib(n):  
    first = 0  
    second = 1  
  
    for index in range(n):  
        yield first  
  
        first, second = second, first*second  
  
print(*fib(3))  
print(*fib(5))
```



Wait, what's this?!?

```
print(*fib(3))
```

- “fib” is a generator function
- When we call it, we get back a generator
- Generators implement the iterator protocol
- Using * before an iterable value in a function call turns all of its values into positional arguments
- (This is related to, but not the same as, *args as a function parameter.)
- “Print” takes any number of positional arguments
- Don't do this with an infinite generator!

Back to our generator...

```
def fib(n):  
    first = 0  
    second = 1  
  
    for index in range(n):  
        yield first  
  
        first, second = second, first*second  
  
print(*fib(3))           # 0 1 0  
print(*fib(5))           # 0 1 0 0 0
```

Fibonacci sequence

Article [Talk](#)

From Wikipedia, the free encyclopedia

For the chamber ensemble, see [Fibonacci Sequence \(ensemble\)](#).

In mathematics, the **Fibonacci sequence** is a [sequence](#) in which each number is the sum of the two preceding ones. Numbers that are part of the Fibonacci sequence are known as **Fibonacci numbers**, commonly denoted F_n . The sequence commonly starts from 0 and 1, but some authors start the sequence from 1 and 1 or sometimes (as did Fibonacci) from 1 and 2. Starting from 0 and 1, the first few values in the sequence are:^[1]

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144.

Concepts similar to Fibonacci numbers were first described in [Indian mathematics](#),^{[2][3][4]} as early as 200 BC in work by [Pingala](#) on enumerating possible patterns of [Sanskrit](#) poetry formed from syllables of two lengths. They are named after the Italian mathematician Leonardo of Pisa, also known as [Fibonacci](#), who introduced the sequence to Western European mathematics in his 1202 book *[Liber Abaci](#)*.^[5]

Fibonacci numbers appear unexpectedly often in mathematics, so much so that there is an entire journal dedicated to their study, the *[Fibonacci Quarterly](#)*. Applications of Fibonacci numbers include computer algorithms such as the [Fibonacci search](#).

[golden ratio](#)

[identities](#)

[action](#)

[is](#)

[visibility](#)

[s](#)

Uh, oh. We have a bug.

Hooray, we have a bug!

Now what?

Option 1: Debugger

- Advantages:
 - Get deep into the code
 - Set breakpoints
- Disadvantages:
 - You have (re-)learn how to use the debugger
 - Slows things down
 - Interferes with the normal running and display
 - You can't (easily, normally) use a debugger in production

Option 2: PySnooper

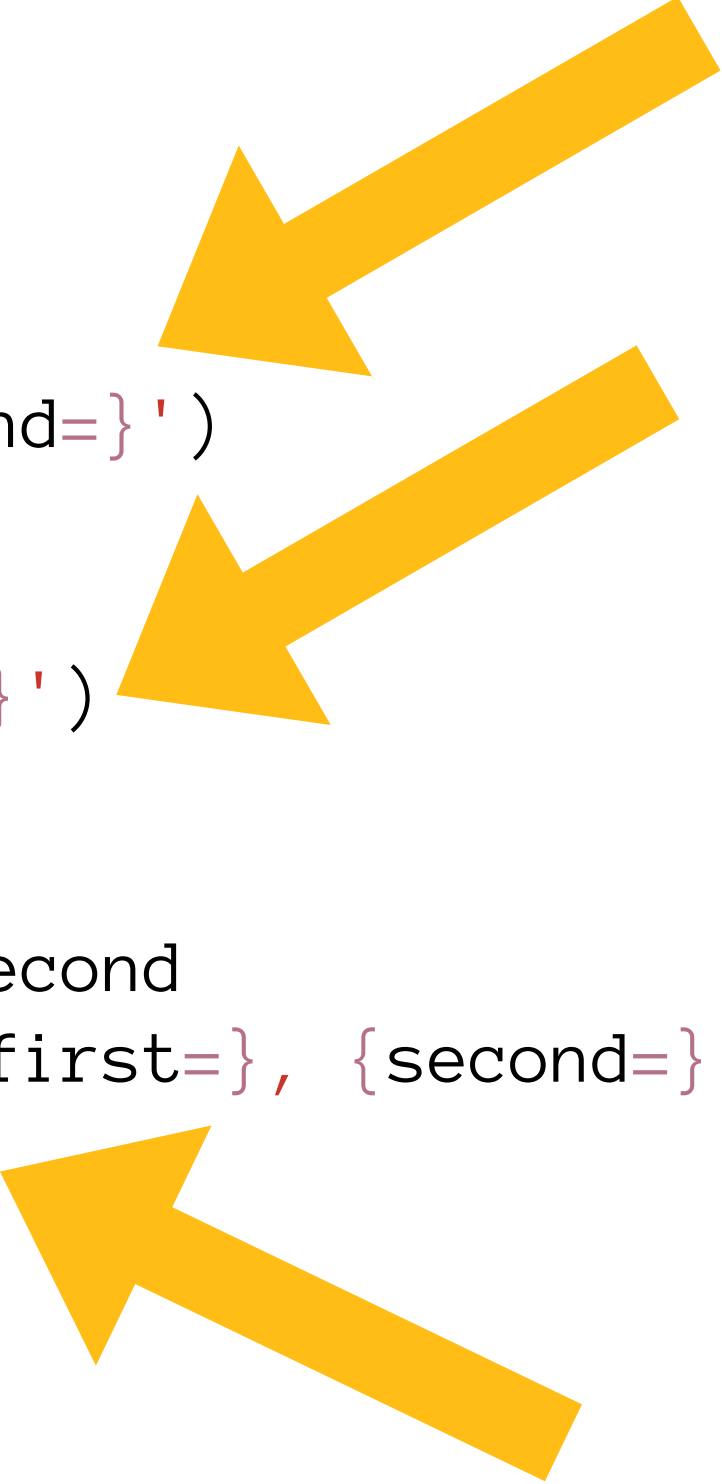
- Advantages:
 - An amazing tool for debugging, from Ram Rachum
- Disadvantages:
 - Displays variables and timing, nothing more
 - Slows things down
 - Interferes with the normal running and display
 - Again, you probably don't want to use it in production

Option 3: print

- This is what everyone reaches for
- After all, it's easy to use and runs fast
- Indeed, for a *long* time, I would use print
 - Unsure of a variable's value? Print it!
 - Not sure how often a function is being called? Print!
 - Want to double check a value's type? Print!
 - What to conditionally display something? Print!

Let's use print

```
def fib(n):  
    first = 1  
    second = 2  
  
    print(f'\t\tAt top, {first=}, {second=}')  
  
    for index in range(n):  
        print(f'\t\tAbout to yield {first}')  
        yield first  
  
        first, second = second, first*second  
        print(f'\t\tAfter assignment, {first=}, {second=}')
```



Printing works!

At top, first=0, second=1

About to yield 0

After assignment, first=1, second=0

About to yield 1

After assignment, first=0, second=0

About to yield 0

After assignment, first=0, second=0

0 1 0

At top, first=0, second=1

About to yield 0

After assignment, first=1, second=0

About to yield 1

After assignment, first=0, second=0

About to yield 0

After assignment, first=0, second=0

About to yield 0

After assignment, first=0, second=0

About to yield 0

After assignment, first=0, second=0

0 1 0 0 0

Print is addictive!

- “print” is intuitive
- You can throw anything at it
- Add f-strings, end and sep, and formatting is easy!
 - (You can even use Rich, and make it look cool)

Print is awful!

- What if we want our output to go to a file, not the screen?
- Do we want these printouts to be displayed in production?
- What if we want to print some things, but not others?
- What if we want to format our output in a uniform way?
- Moreover: DRY up these decisions in a single place

You want the “logging” module

Why use logging?

- It's part of the standard library
- It solves all of the problems mentioned earlier
- For simple things, it's just a tiny bit harder than “print”
- For complex things and production code, it's far superior

Starting off

- Let's do a simple “hello, world” of logging:

```
import logging
logging.error( 'Hello, world! ' )
```

- First, import the “logging” module
- Then, invoke the “error” method to write to the log

Screen output has three parts

(2) Logger used; default is “root”

ERROR:root:Hello, world!

(1) Log level

(3) Log message

Log levels

- Not all messages are equally important
 - Sometimes we want to see only the most important ones
 - Other times, we want to see everything
- For this reason, there are five different *log levels*:
 - critical (highest priority)
 - error
 - warning
 - info
 - debug (lowest priority)

One function per log level

- The “logging” module defines one function for each log level
- When we invoke that function, a message is logged at that level

```
import logging
logging.critical( 'Hello, world!' )
logging.error( 'Hello, world!' )
logging.warning( 'Hello, world!' )
logging.info( 'Hello, world!' )
logging.debug( 'Hello, world!' )
```

Let's run that code!

```
CRITICAL:root:Hello, world!
```

```
ERROR:root:Hello, world!
```

```
WARNING:root:Hello, world!
```

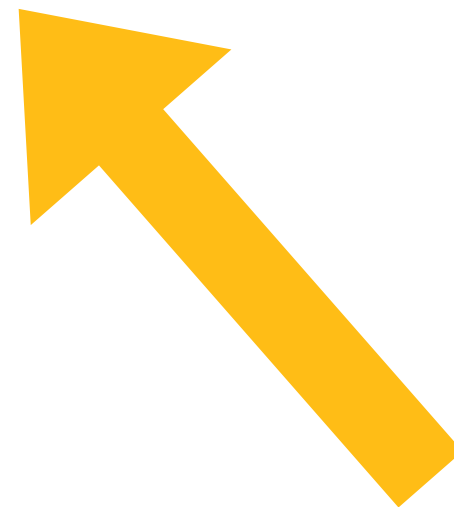
- Where did the “info” and “debug” messages go?
- By default, we only see messages of level “warning” and above
- We can change this by configuring the module to show everything from level “debug” and above

Simple configuration

- We invoke “logging.basicConfig” to specify which levels to show

```
logging.basicConfig(level=logging.DEBUG)
```

- Notice:
 - The functions are in lowercase
 - The log levels are CAPITALIZED



Already, we're better off!

- In your code:
 - Use `logging.error` and `logging.critical` for important stuff
 - Use `logging.warning`, `logging.info`, and `logging.debug` for less crucial things
- Then, adjust your configuration:
 - When developing, set the minimum to `logging.INFO` or `logging.DEBUG`
 - When replying, set the minimum to `logging.WARNING` or `logging.ERROR`

Standard error

- When we use “print”, output normally goes to “standard output”
 - In other words, the screen
 - Unless you’ve redirected output elsewhere
- When we use logging, output goes to “standard error”
 - Not redirected along with standard output
 - Normally displayed on the screen no matter what
 - Can be redirected separately

More interesting output

- We'll normally want to log one or more variable values
- Sounds like a job for f-strings, right?

```
import logging
logging.basicConfig(level=logging.DEBUG)

name = input('Enter your name: ').strip()

logging.info(f'User entered {name=}')
print(f'Hello, {name}')
```



Unfortunately, this works.

Don't use f-strings!

- I love f-strings. Really, I do.
- But it's recommended that we *not* use f-strings when logging:
 - f-strings are evaluated when they are created, not when they're used
 - Meaning: They're evaluated before we decide whether to log
 - Moreover: f-strings can be used for denial-of-service attacks with untrusted variables
- These aren't small things — particularly the potential for DOS


Instead, use %

- Similar to %-style interpolation

```
import logging
logging.basicConfig(level=logging.DEBUG)

name = input('Enter your name: ')

logging.info('User entered %s', name)
print(f'Hello, {name}')
```



Multiple loggers

- We can define multiple loggers, each with its own configuration
- This way, each part of our program can have its own priorities, and its own logging destinations:
 - Backend
 - UI
 - Algorithms
 - Devops
- Use “logging.getLogger” to create/retrieve a logger
 - If one already exists with the specified string, we get the existing logger

Example of multiple loggers

```
import logging
```

```
logging.basicConfig(level=logging.DEBUG)
```

```
backend_logger = logging.getLogger('backend')  
database_logger = logging.getLogger('database')
```

```
x = 100
```

```
backend_logger.error('problem with backend')  
backend_logger.debug('x = %s', x)
```

```
database_logger.error('problem with database')  
database_logger.debug('x = %s', x)
```

Output from multiple loggers

ERROR:backend:problem with backend

DEBUG:backend:x = 100

ERROR:database:problem with database

DEBUG:database:x = 100

Writing to a file

- Writing to a file is even better than writing to stderr!
- Pass the “filename” keyword argument to basicConfig

```
import logging
```

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

```
backend_logger = logging.getLogger('backend')  
database_logger = logging.getLogger('database')
```



```
x = 100
```

```
backend_logger.error('problem with backend')  
backend_logger.debug('x = %s', x)
```

```
database_logger.error('problem with database')  
database_logger.debug('x = %s', x)
```

Handlers

- The good news? All output now goes to a file, not stderr
- The bad news? *All* output now goes to the same file
- So much for distinguishing multiple loggers, right?
- How and where we write is determined by *handlers*
 - By default, we use a “StreamHandler” to stderr
 - You can change it to stdout, if you really want...
 - Passing filename to basicConfig uses a “FileHandler”
 - Each logger can have one or more handlers, independent of other loggers

Get rid of basicConfig

- It's easy to work with!
- But it also limits us, and the configuration options we can pass
- We can create the logger, then add various parts to its configuration with method calls

Two loggers, two handlers

```
import logging
```

```
backend_logger = logging.getLogger('backend')  
backend_logger.setLevel(logging.ERROR)  
backend_logger.addHandler(logging.FileHandler('mylog.txt'))  
)
```

```
database_logger = logging.getLogger('database')  
database_logger.setLevel(logging.DEBUG)  
database_logger.addHandler(logging.StreamHandler())
```

```
x = 100
```

```
backend_logger.error('problem with backend') # to file  
backend_logger.debug('x = %s', x)           # ignored
```

```
database_logger.error('problem with database') # to screen  
database_logger.debug('x = %s', x)             # to screen
```

Two loggers, one handler

```
import logging

backend_logger = logging.getLogger('backend')
fh = logging.FileHandler('mylog.txt')

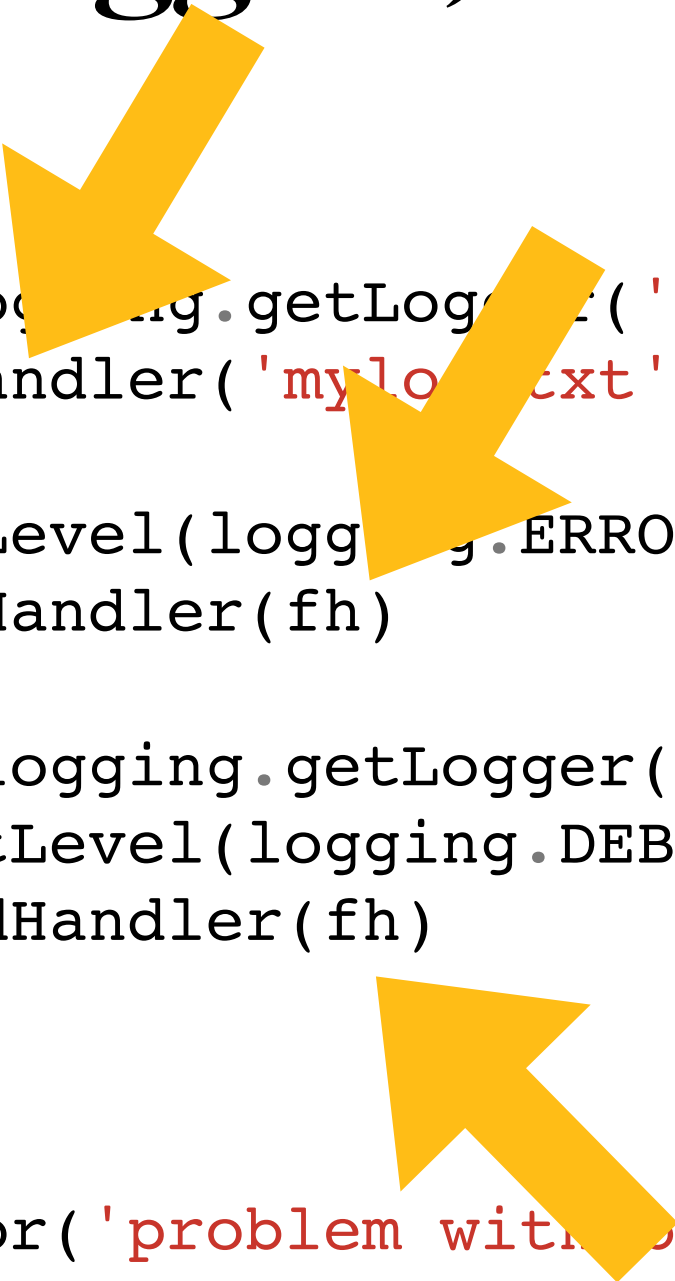
backend_logger.setLevel(logging.ERROR)
backend_logger.addHandler(fh)

database_logger = logging.getLogger('database')
database_logger.setLevel(logging.DEBUG)
database_logger.addHandler(fh)

x = 100

backend_logger.error('problem with backend')
backend_logger.debug('x = %s', x)

database_logger.error('problem with database')
database_logger.debug('x = %s', x)
```



Two loggers, three handlers

```
import logging

backend_logger = logging.getLogger('backend')
fh = logging.FileHandler('mylog.log')
bh = logging.FileHandler('backend.txt')

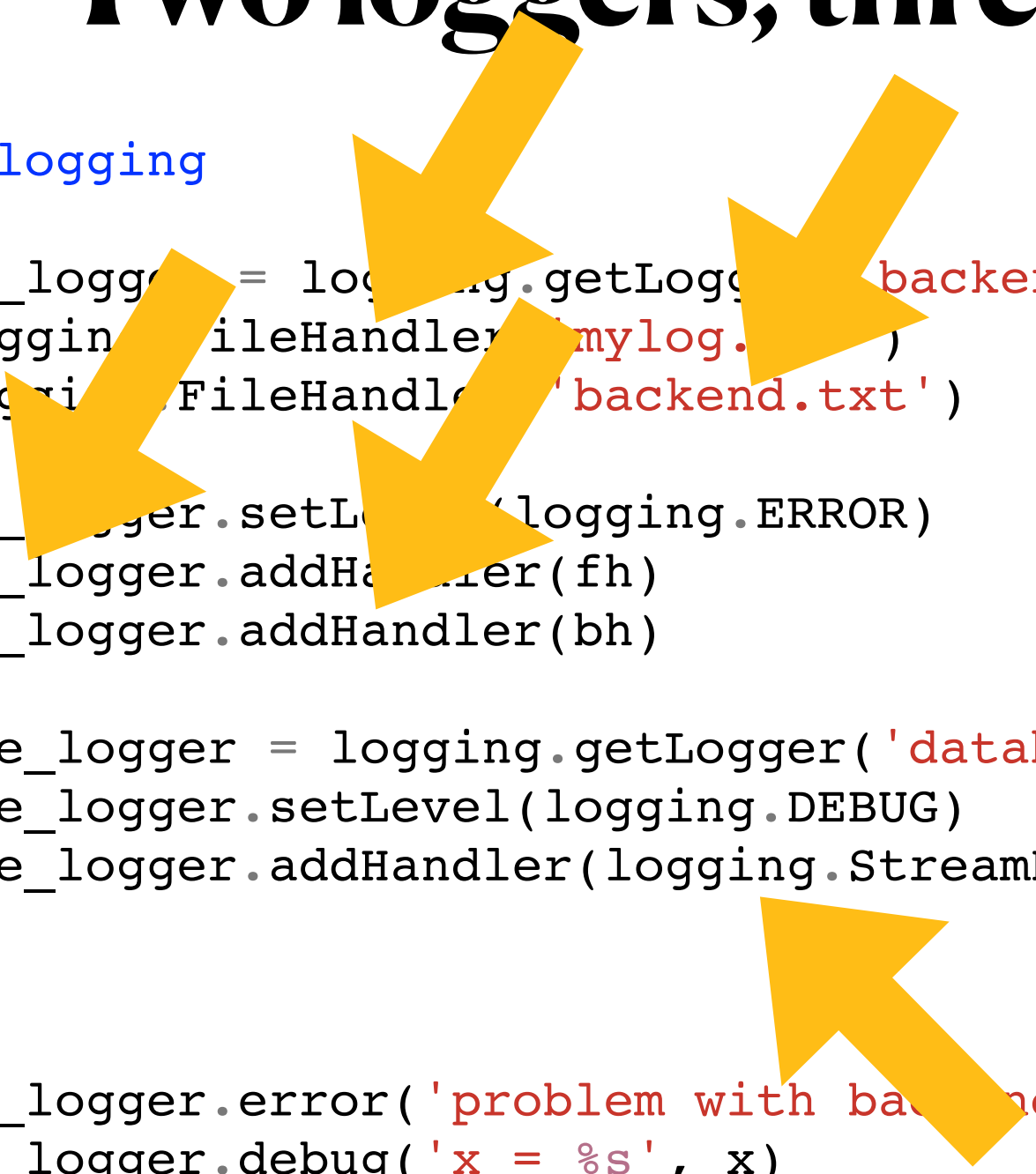
backend_logger.setLevel(logging.ERROR)
backend_logger.addHandler(fh)
backend_logger.addHandler(bh)

database_logger = logging.getLogger('database')
database_logger.setLevel(logging.DEBUG)
database_logger.addHandler(logging.StreamHandler())

x = 100

backend_logger.error('problem with backend')
backend_logger.debug('x = %s', x)

database_logger.error('problem with database')
database_logger.debug('x = %s', x)
```



You can set the level per handler

- We've seen how we can set the minimum level for a logger
- But we can also set the minimum level for a *handler*
- For example: Write important things to the screen, and all things to a log file.

Handlers defined in “logging”

- StreamHandler
- FileHandler
- NullHandler
- WatchedFileHandler
- RotatingFileHandler
- TimedRotatingFileHandler
- SocketHandler
- DatagramHandler
- SysLogHandler
- NTEventLogHandler
- SMTPHandler
- MemoryHandler
- HTTPHandler
- QueueHandler

Formatters

- We can also customize how our messages are written
- This is done via objects called *Formatters*
- Create a new instance of `logging.Formatter`, passing a string
- Assign a formatter to a logger with “`setFormatter`”
- The formatter can refer to attributes of the logging object
 - Just use `%(NAME)s` in the string
- Example names: `asctime`, `filename`, `levelname`, `msg`, `process`, `thread`
- You can also create custom formatter classes

1 logger, 2 handlers, 2 formatters

```
import logging

logger = logging.getLogger('mylogger')

f1 = logging.Formatter('%(asctime)s] %(levelname)s\t%(message)s')
f2 = logging.Formatter('\n\n\nPAY ATTENTION!\n\n\n[%(asctime)s] %(levelname)s\t%(message)s\n\n\n')

fh = logging.FileHandler('mylog.txt')
fh.setLevel(logging.INFO)
fh.setFormatter(f1)
logger.addHandler(fh)

sh = logging.StreamHandler()
sh.setLevel(logging.CRITICAL)
sh.setFormatter(f2)
logger.addHandler(sh)

name = input('Enter your name: ').strip()
logger.info('[info] Got name %s', name)
logger.critical('[critical] Got name %s', name)
```

Three yellow arrows are overlaid on the code. The first arrow points from the top left towards the 'import logging' line. The second arrow points from the middle right towards the 'fh' (FileHandler) configuration block. The third arrow points from the bottom right towards the 'sh' (StreamHandler) configuration block.

Using Rich

- Want to use the Rich package to format your log messages?
- Just use the `rich.logging.RichHandler` as your handler:

```
from rich.logging import RichHandler  
  
fh = RichHandler()
```

- It automatically colorizes the error level and time, with timestamps on the left and line numbers on the right
- You can also use all of Rich's formatting:

```
logger.critical('[yellow]Got name %s[/yellow]',  
                name)
```

Actually, you need a bit more...

- To activate Rich formatting, pass the “extra” keyword argument
- It takes a dict value; add ‘markup’=True to add formatting

```
logger.critical( '[yellow]Got name %s[/yellow]',  
                 name,  
                 extra={ 'markup': True } )
```

Date formatting

- We often create a Formatter passing a message-format string
- But we can also pass a second (optional) argument, a *date* formatting string
- This uses the same %-codes as strftime, so you can display things in all sorts of weird and wild ways!

Filters

- Often, we can use an “if” to determine if we want to log
- But how much filtering logic do we want in our main code?
- We can put such logic in a filter:
 - Define a class
 - Define a method in the class that takes two arguments, self and record
 - The record will contain all of the fields from the log message
 - If the function returns True, the message is logged
 - If the function returns False, no message is logged

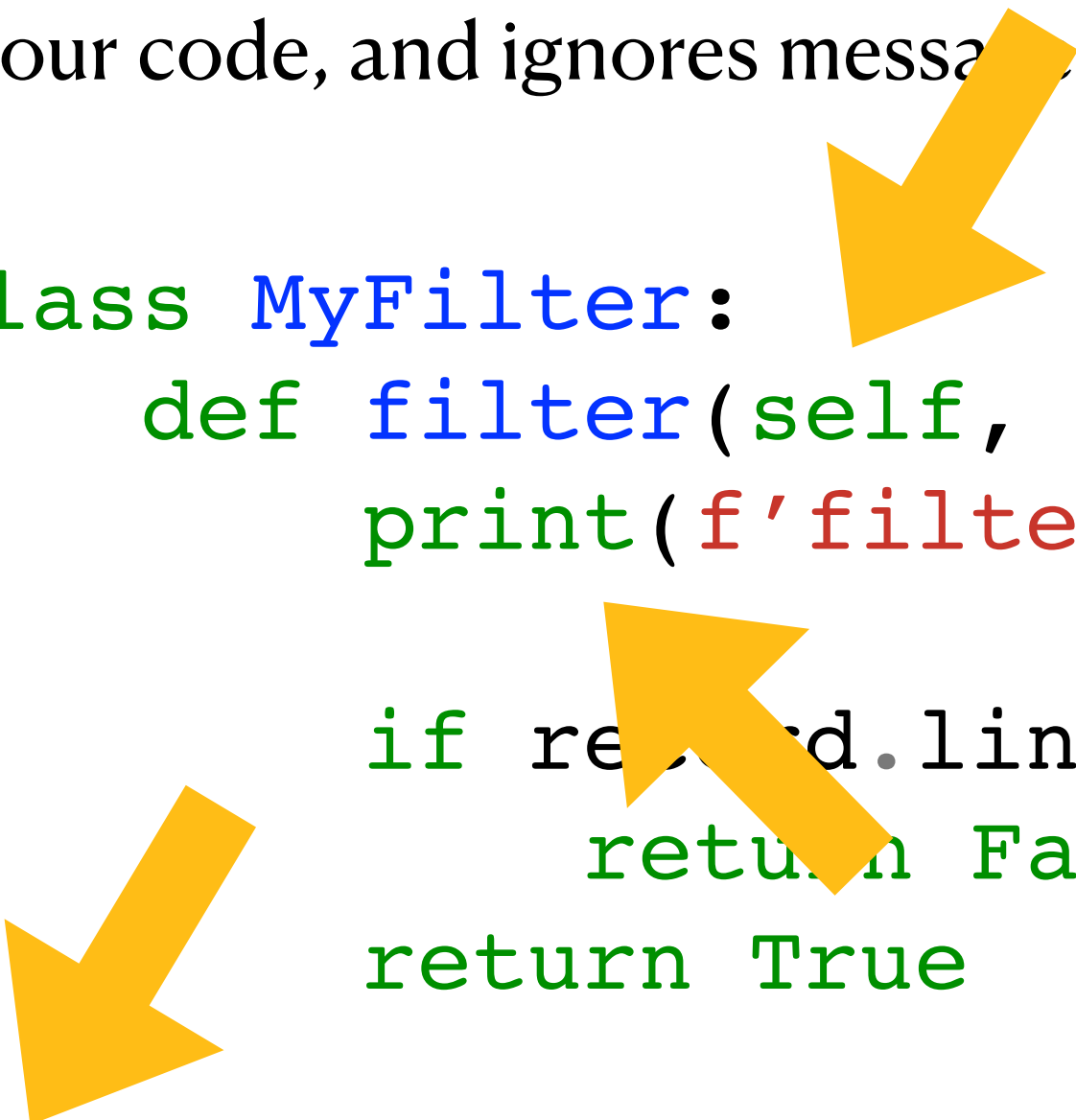
Simple filter

- Here's a filter that logs messages from odd-numbered lines in our code, and ignores messages from even-numbered ones:

```
class MyFilter:
    def filter(self, record):
        print(f'filter: {vars(record)}')

        if record.lineno % 2:
            return False
        return True

fh.addFilter(MyFilter())
```



Filter with a function!

```
def myfilter(record):  
    print(f'filter: {vars(record)}')  
  
    if record.lineno % 2:  
        return False  
    return True  
  
fh.addFilter(myfilter)
```

Configuration

- So far, our configuration has been piecemeal:
 - Create a logger
 - Call a bunch of methods to set log levels, handlers, formatters, and filters
- We can instead define our configuration with a dict
- Then we can load it all at once!

```

log_dict = {
    'version': 1,
    'formatters': {
        'short': {
            'format': '[%(asctime)s] %(levelname)s\t%(message)s'
        },
        'long': {
            'format': '\n\n\nPAY ATTENTION!\n\n\n[%(asctime)s] %(levelname)s\t%(message)s\n\n\n'
        }
    },

    'handlers': {
        'fh': {
            'class': 'logging.FileHandler',
            'formatter': 'short',
            'filename': 'mylog.txt',
            'level': logging.INFO
        },
        'sh': {
            'class': 'logging.StreamHandler',
            'formatter': 'long',
            'level': logging.CRITICAL
        }
    },

    'loggers':
    {
        'mylogger': {
            'handlers': ['fh', 'sh'],
            'level': logging.INFO
        }
    }
}

```

Loading our config dict

- With our dict defined, we can now use it for configuration:

```
logger = logging.getLogger( 'mylogger' )  
logging.config.dictConfig(log_dict)
```

- We can also define it in a module, using the dict across programs

Logging exceptions


- We often want to log exceptions, along with some context
- The “`logger.exception`” function does this
 - It is meant to be used within an “except” clause
 - It adds exception information, for easier understanding

Logging exceptions

```
import logging
```

```
logger = logging.getLogger( 'mylogger' )  
logging.basicConfig(level=logging.DEBUG,  
                    filename='zerolog.txt' )
```

```
try:  
    print(100/0)  
except ZeroDivisionError as e:  
    logger.exception( 'Avoid dividing by  
zero' )
```



What is logged?

```
ERROR:mylogger:Avoid dividing by zero
```

```
Traceback (most recent call last):
```

```
  File "/Users/reuven/Conferences/EuroPython/2023/  
logging-code/./code18.py", line 10, in <module>
```

```
    print(100/0)
```

```
~~~~^~
```

```
ZeroDivisionError: division by zero
```


Development vs. production

- It's nice that we can configure different log levels
- But do we really want to be modifying our code when we switch to development vs. production?
- Better: Use an environment variable to influence the code
-

Using an environment variable



```
import os
```

```
if os.getenv( 'DEVELOPMENT_MODE' ):  
    logging.basicConfig(level=logging.DEBUG)  
    logging.debug( 'Development mode!' )  
else:  
    logging.basicConfig(level=logging.ERROR)
```

Debugging our “fib” generator

```
import logging
import os

if os.getenv('DEVELOPMENT_MODE'):
    logging.basicConfig(level=logging.DEBUG)
    logging.debug('Running in development mode!')
else:
    logging.basicConfig(level=logging.ERROR)

def fib(n):
    first = 0
    second = 1

    for index in range(n):
        yield first

        logging.debug('Before, first = %s, second = %s', first, second)
        first, second = second, first*second
        logging.debug('\tAfter, first = %s, second = %s', first, second)
```

When I run it...

```
[ ~/Downloads]$ DEVELOPMENT_MODE=1 ~/Desktop/code02.py
```

```
DEBUG:root:Running in development mode!
```

```
DEBUG:root:Before, first = 0, second = 1
```

```
DEBUG:root:    After, first = 1, second = 0
```

```
DEBUG:root:Before, first = 1, second = 0
```

```
DEBUG:root:    After, first = 0, second = 0
```

```
DEBUG:root:Before, first = 0, second = 0
```

```
DEBUG:root:    After, first = 0, second = 0
```

```
0 1 0
```

```
DEBUG:root:Before, first = 0, second = 1
```

```
DEBUG:root:    After, first = 1, second = 0
```

```
DEBUG:root:Before, first = 1, second = 0
```

```
DEBUG:root:    After, first = 0, second = 0
```

```
DEBUG:root:Before, first = 0, second = 0
```

```
DEBUG:root:    After, first = 0, second = 0
```

```
DEBUG:root:Before, first = 0, second = 0
```

```
DEBUG:root:    After, first = 0, second = 0
```

```
DEBUG:root:Before, first = 0, second = 0
```

```
DEBUG:root:    After, first = 0, second = 0
```

```
0 1 0 0 0
```

Summary

- Yes, it's quick and easy to use “print”
- But with a tiny bit of additional code, you can get *much* more power:
 - Choose what you want to write
 - Where you want to write it
 - How you want to format it
 - Under what conditions it should be written
 - Write it to more than one location
- Best of all, the configuration is centralized — so moving from development to production is easy.

Resources

- The logging documentation on [python.org](https://docs.python.org/3.11/howto/logging-cookbook.html) is great
- In particular, check out the “logging cookbook,” which covers a very large number of use cases:
- <https://docs.python.org/3.11/howto/logging-cookbook.html>

Questions?

- Contact me:
 - reuven@lerner.co.il
 - <https://lerner.co.il>
 - Find me on Twitter, Threads, and YouTube!
- Python newsletter: <https://BetterDevelopersWeekly.com>
- Pandas puzzles: <https://BambooWeekly.com>

