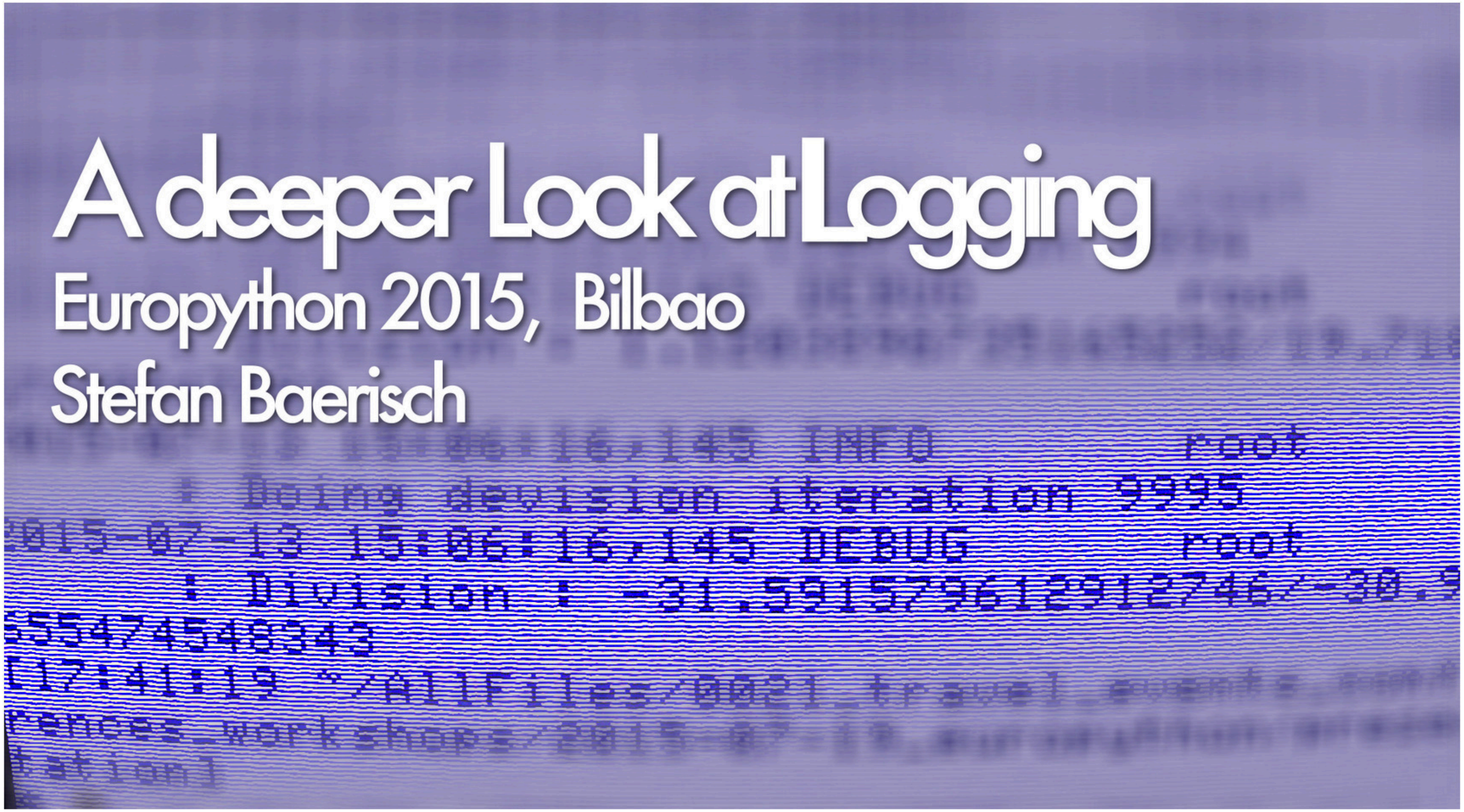
A deeper Look at Logging

Europython 2015, Bilbao

Stefan Baerisch



```
                     16,145 INFO               root
       : Doing devision iteration 9995
2015-07-13 15:06:16,145 DEBUG              root
       : Division : -31.591579612912746/-30.9
555474548343
[17:41:19 ~/AllFiles/0021_travel_events_
rences_workshops/2015-07-13_europython_prese
ntation]
```

Agenda

- Why Logging
- How does Logging work for you?
- Optional Content

The Presentation

- The slides, support code and jypyter notebook are on Github
- https://github.com/stbaercom/europython2015_logging

A Simple Program, Without any Logging

```
from datetime import datetime

def my_division_p(dividend, divisor):
    try:
        print("Debug, Division : {}/{}".format(dividend,divisor))
        result = dividend / divisor
        return result
    except (ZeroDivisionError, TypeError):
        print("Error, Division Failed")
        return None
def division_task_handler_p(task):
    print("Handling division task,{} items".format(len(task)))
    result = []
    for i, task in enumerate(task):
        print("Doing devision iteration {} on {:%Y}".format(i,datetime.now()))
        dividend, divisor = task
        result.append(my_division_p(dividend,divisor))
    return result
```

Let us Have a Look at the Output

```
task = [(3,4),(5,1.4),(2,0),(3,5),("10",1)]
division_task_handler_p(task)
```

```
Handling division task,5 items
Doing devision iteration 0 on 2015
Debug, Division : 3/4
Doing devision iteration 1 on 2015
Debug, Division : 5/1.4
Doing devision iteration 2 on 2015
Debug, Division : 2/0
Error, Division Failed
Doing devision iteration 3 on 2015
Debug, Division : 3/5
Doing devision iteration 4 on 2015
Debug, Division : 10/1
Error, Division Failed

 [0.75, 3.5714285714285716, None, 0.6, None]
```

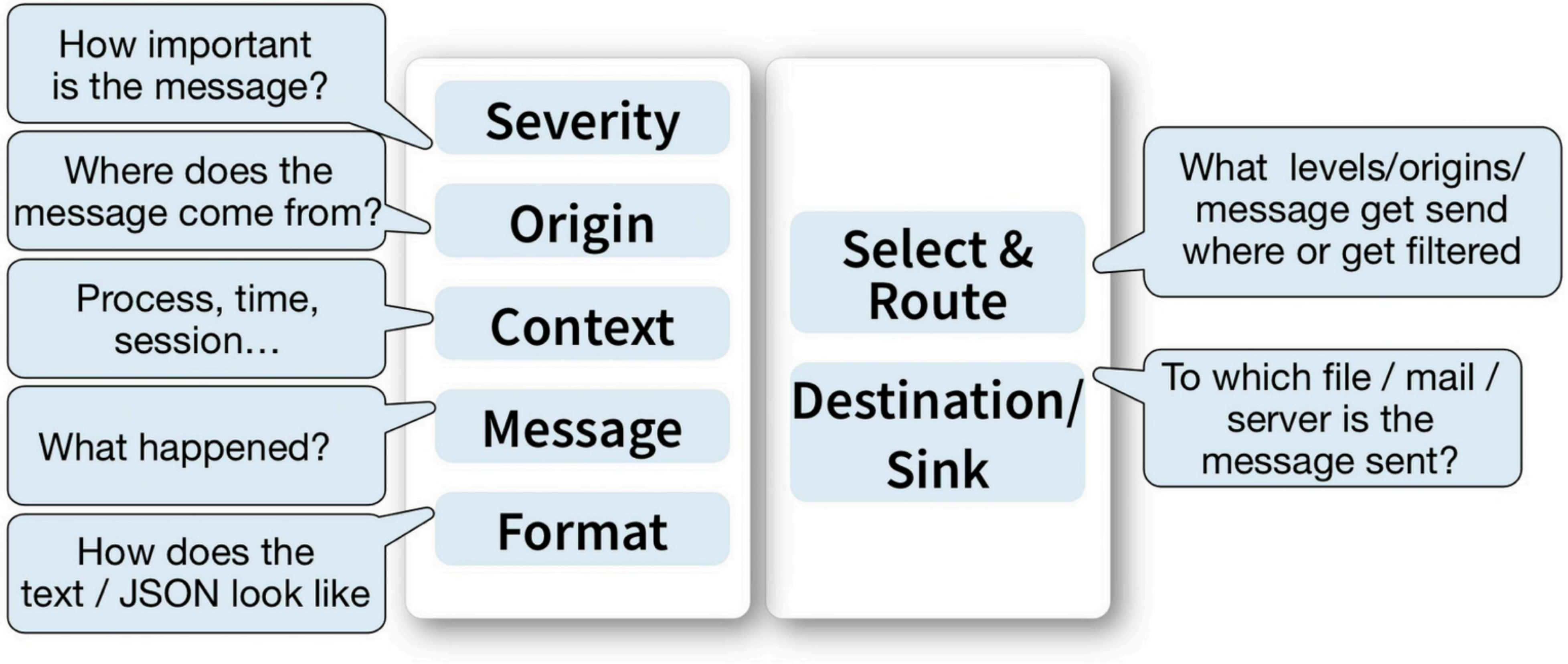
The Problems with `print()`

- We don't have a way to select the types of messages we are interested in
- We have to add all information (timestamps, etc...) by ourselves
- All our messages will look slightly different
- We have only limited control where our message end up

What is Different with Logging?

- We have more structure, and easier parsing
- The logging module provides some extra informaiton (Logger, Level, and Formating)
- We Handling of exception essentially for free.

Aspects of a Logging Message


How important is the message?
Severity
Where does the message come from?
Origin
Process, time, session…
Context
What happened?
Message
How does the text / JSON look like
Format
Select & Route
What levels/origins/ message get send where or get filtered
Destination/ Sink
To which file / mail / server is the message sent?

Using the Logging Module for Comparison

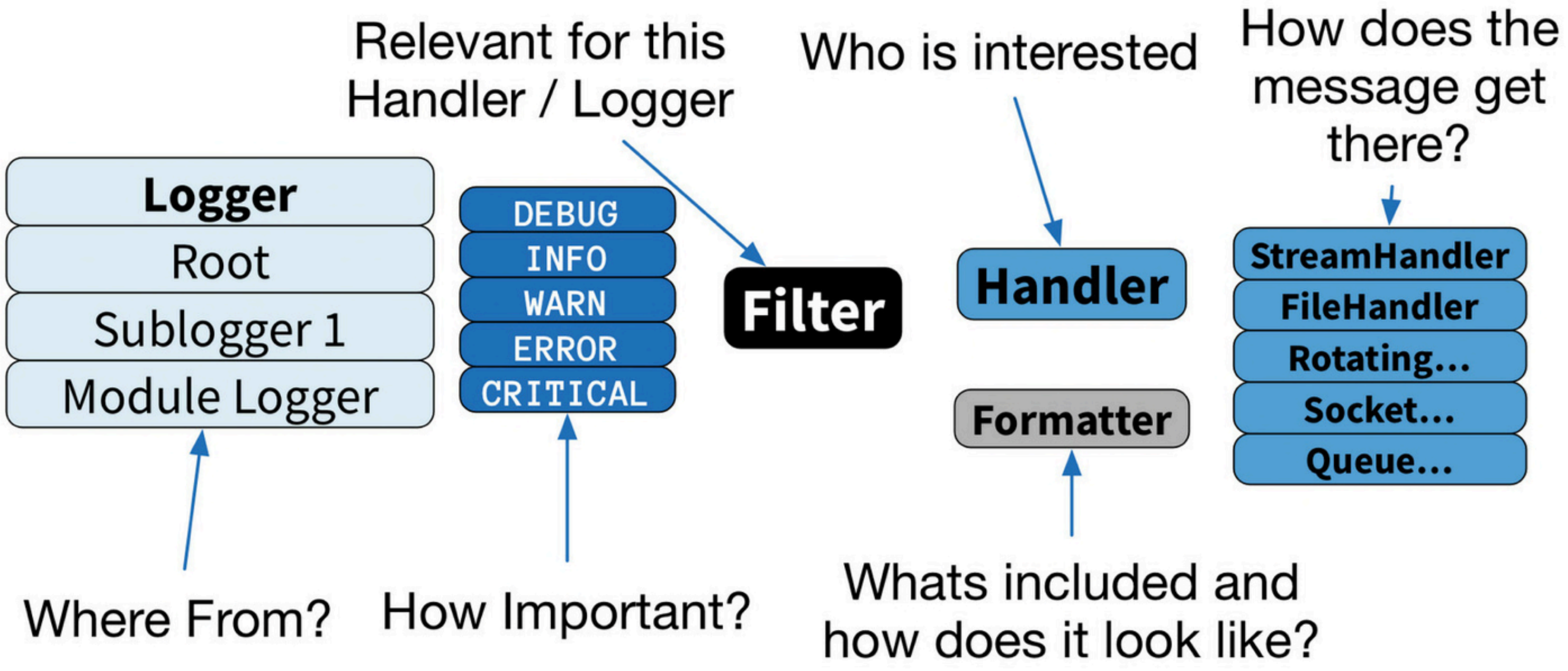
```python
import log1; logging = log1.get_clean_logging()
logging.basicConfig(level=logging.DEBUG)
log = logging.getLogger()

def my_division(dividend, divisor):
    try:
        log.debug("Division : %s/%s", dividend, divisor)
        result = dividend / divisor
        return result
    except (ZeroDivisionError, TypeError):
        log.exception("Error, Division Failed")
        return None

def division_task_handler(task):
    log.info("Handling division task,%s items",len(task))
    result = []
    for i, task in enumerate(task):
        log.info("Doing devision iteration %s",i)
        dividend, divisor = task
        result.append(my_division(dividend,divisor))
    return result
```

The Call and the Log Messages

```python
task = [(3,4),(2,0),(3,5),("10",1)]
division_task_handler(task)
```

```
INFO:root:Handling division task,4 items
INFO:root:Doing devision iteration 0
DEBUG:root:Division : 3/4
INFO:root:Doing devision iteration 1
DEBUG:root:Division : 2/0
ERROR:root:Error, Division Failed
Traceback (most recent call last):
  File "<ipython-input-10-a904db1e3e23>", line 8, in my_division
    result = dividend / divisor
ZeroDivisionError: division by zero
INFO:root:Doing devision iteration 2
DEBUG:root:Division : 3/5
INFO:root:Doing devision iteration 3
DEBUG:root:Division : 10/1
ERROR:root:Error, Division Failed
Traceback (most recent call last):
  File "<ipython-input-10-a904db1e3e23>", line 8, in my_division
    result = dividend / divisor
TypeError: unsupported operand type(s) for /: 'str' and 'int'

 [0.75, None, 0.6, None]
```

How does the Logging Module represent these Aspect



Back to Code. How does Logging Work?

```python
import log1;logging = log1.get_clean_logging() # this would be import logging outside this notebook

logging.debug("Find me in the log")
logging.info("I am hidden")
logging.warn("I am here")
logging.error("As am I")
try:
    1/0;
except:
    logging.exception(" And I")
logging.critical("Me, of course")
```

```
WARNING:root:I am here
ERROR:root:As am I
ERROR:root: And I
Traceback (most recent call last):
  File "<ipython-input-12-75f8227eec02>", line 8, in <module>
    1/0;
ZeroDivisionError: division by zero
CRITICAL:root:Me, of course
```

More Complex Logging Setup with `basicConfig()`

```python
import log1;logging = log1.get_clean_logging()

datefmt = "%Y-%m-%d %H:%M:%S"
msgfmt = "%(asctime)s,%(msecs)03d %(levelname)-10s %(name)-15s : %(message)s"
logging.basicConfig(level=logging.DEBUG, format=msgfmt, datefmt=datefmt)
logging.debug("Now I show up ")
logging.info("Now this is %s logging!","good")
logging.warn("I am here. %-4i + %-4i = %i",1,3,1+3)
logging.error("As am I")
try:
    1/0;
except:
    logging.exception(" And I")
```

```
2015-07-19 20:19:55,551 DEBUG      root            : Now I show up
2015-07-19 20:19:55,552 INFO       root            : Now this is good logging!
2015-07-19 20:19:55,552 WARNING    root            : I am here. 1    + 3    = 4
2015-07-19 20:19:55,552 ERROR      root            : As am I
2015-07-19 20:19:55,553 ERROR      root            :  And I
Traceback (most recent call last):
  File "<ipython-input-13-63765f2f7e9f>", line 12, in <module>
    1/0;
ZeroDivisionError: division by zero
```

Some (personal) Remarks about `basicConfig()`

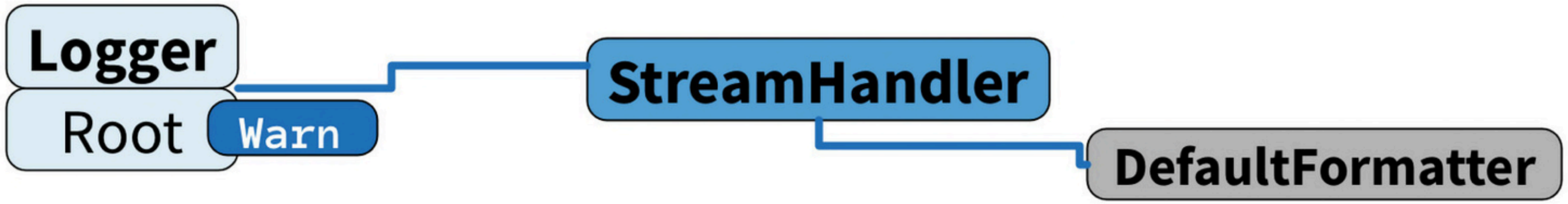
- `basicConfig()` does save you some typing, but I would go for the 'normal' setup.
- Using `basicConfig()` is a matter of personal taste.
- The normal setup makes the structure clearer.
- Keep in mind that basicConfig() is meant to be called once...

Using the Standard Configuration

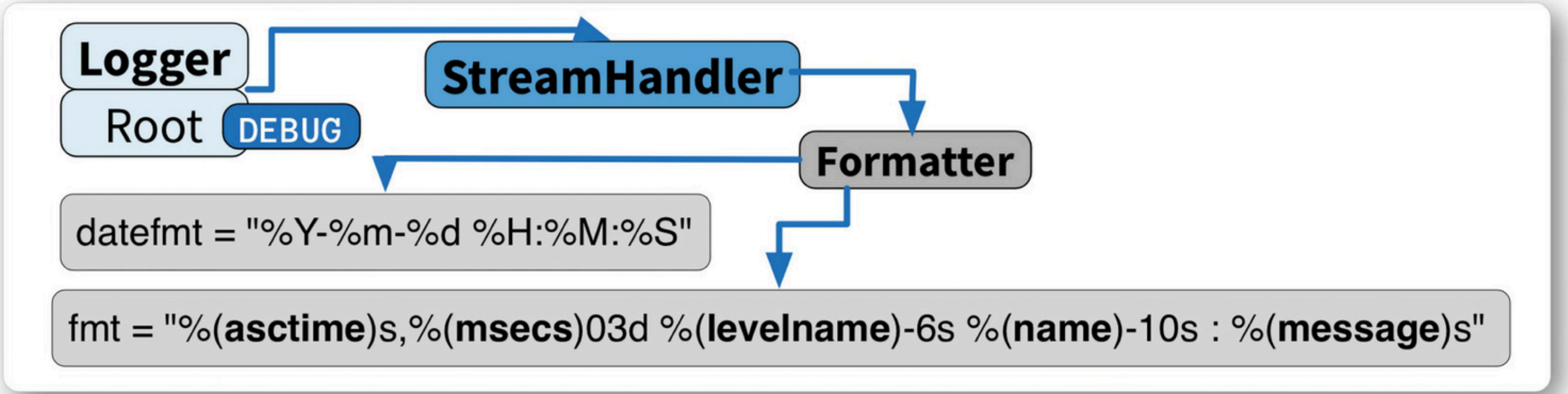
```python
import log1, json, logging.config;logging  = log1.get_clean_logging()
datefmt = "%Y-%m-%d %H:%M:%S"
msgfmt = "%(asctime)s,%(msecs)03d %(levelname)-6s %(name)-10s : %(message)s"

log = logging.getLogger()
log.setLevel(logging.DEBUG)
lh = logging.StreamHandler()
lf =  logging.Formatter(fmt=msgfmt, datefmt=datefmt)
lh.setFormatter(lf)
log.addHandler(lh)

log.info("Now this is %s logging!","good")
log.debug("A slightly more complex message %s + %s = %s",1,2,1+2)
```

```
2015-07-19 20:19:55,571 INFO   root       : Now this is good logging!
2015-07-19 20:19:55,572 DEBUG  root       : A slightly more complex message 1 + 2
= 3
```

Now, back to the Theory. What have we Build?



How do we get from the Configuration to the Log Message?



datefmt = "%Y-%m-%d %H:%M:%S"

fmt = "%(**asctime**)s,%(**msecs**)03d %(**levelname**)-6s %(**name**)-10s : %(**message**)s"

```
log.info( "A slightly more complex message %s + %s = %s" , 1+2 , 2 , 1 )
```

```
2015-07-14 10:42:34,379 DEBUG root : A slightly more complex
message 1 + 2 = 3
```

Formatting : Attributes Available for the Logging Call

Attribute	Description
args	Tuple of arguments passed to the logging call
asctime	Log record creation time, formatted
created	Log record creation time, seconds since the Epoch
exc_info	Exception information / stack trace, if any
filename	Filename portion of pathname for the logging module
funcName	Name of function containing the logging call
levelname	Name of Logging Level
levelno	Number of Logging Level
lineno	Line number in source code for the logging call
module	Module (name portion of filename).
message	Logged message
name	Name of the logger used to log the call.
pathname	pathname of source file
process	Process ID
processName	Process name
...	...

Using `dictConfig()`

```
import log1, json, logging.config;logging  = log1.get_clean_logging()
conf_dict = {
    'version': 1,
    'disable_existing_loggers': True,
    'formatters': {
        'longformat': {
            'format': "%(asctime)s,%(msecs)03d %(levelname)-10s %(name)-15s : %(messa
ge)s",
            'datefmt': "%Y-%m-%d %H:%M:%S"}},
    'handlers': {
        'console': {
            'class': 'logging.StreamHandler',
            'formatter': "longformat"}},
    'loggers':{
        '': {
            'level': 'DEBUG',
            'handlers': ['console']}}}
logging.config.dictConfig(conf_dict)
log = logging.getLogger()
log.info("Now this is %s logging!","good")
```

```
2015-07-19 20:19:55,602 INFO       root                : Now this is good logging!
```

Adding a `Filehandler` to the Logger

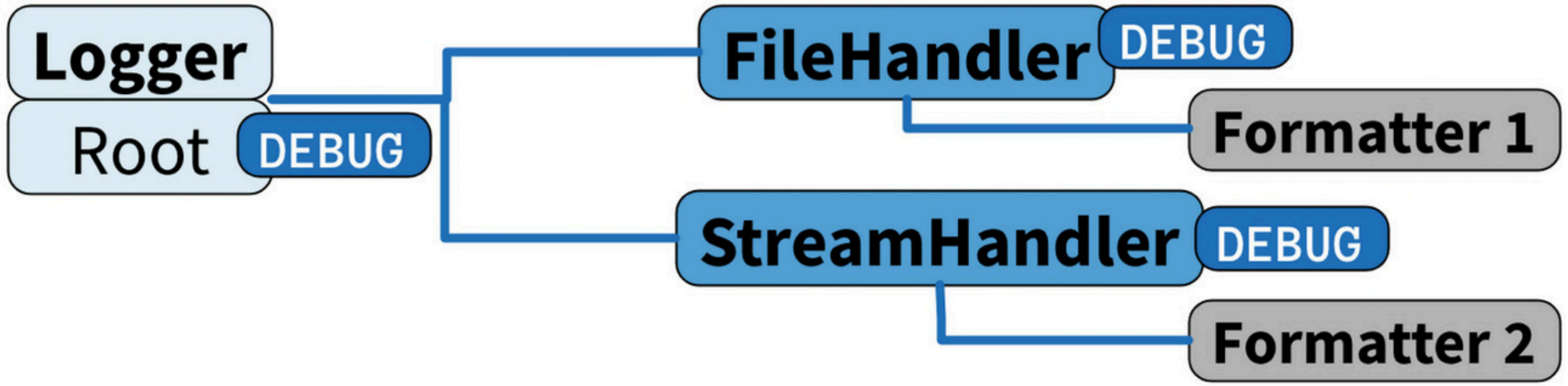
```python
import log1, json, logging.config;logging  = log1.get_clean_logging()
base_config = json.load(open("conf_dict.json"))

base_config['handlers']['logfile'] = {
    'class' : 'logging.FileHandler',
    'mode' : 'w',
    'filename' : 'logfile.txt',
    'formatter': "longformat"}
base_config['loggers']['']['handlers'].append('logfile')
logging.config.dictConfig(base_config)
log = logging.getLogger()
log.info("Now this is %s logging!","good")
!cat logfile.txt
```

```
2015-07-19 20:19:55,618 INFO      root             : Now this is good logging!

2015-07-19 20:19:55,618 INFO      root             : Now this is good logging!
```

StreamHandler

FileHandler

Another look at the logging object tree



Set the Level on the `FileHandler`

```python
import log1, json, logging.config;logging  = log1.get_clean_logging()

file_config = json.load(open("conf_dict_with_file.json"))
file_config['handlers']['logfile']['level'] = "WARN"
logging.config.dictConfig(file_config)
log = logging.getLogger()
log.info("Now this is %s logging!","good")
log.warning("Now this is %s logging!","worrisome")
!cat logfile.txt
```

```
2015-07-20 19:04:03,132 INFO      root             : Now this is good logging!
2015-07-20 19:04:03,133 WARNING   root             : Now this is worrisome loggin
g!

2015-07-20 19:04:03,133 WARNING   root             : Now this is worrisome loggin
g!
```

StreamHandler

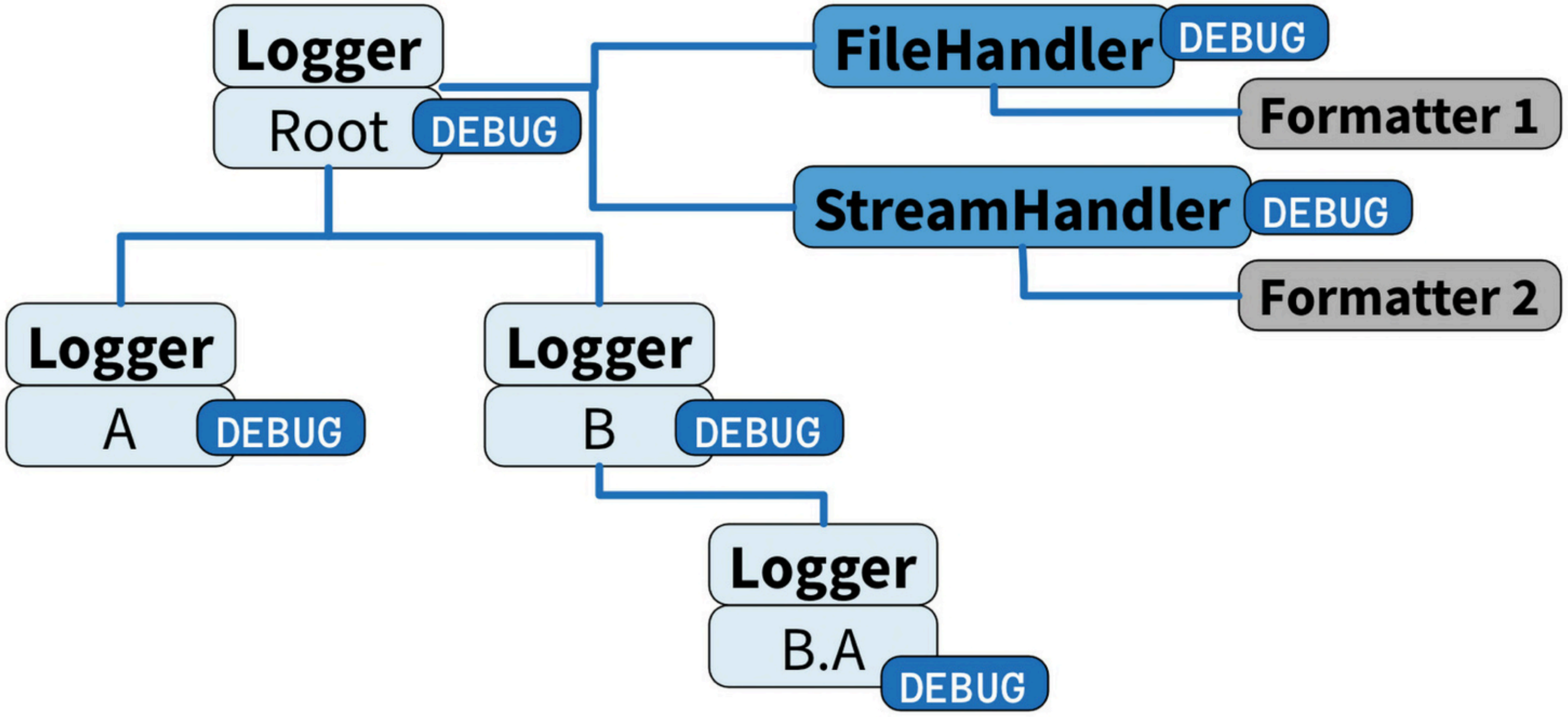
FileHandler

Adding Child Loggers under the Root

```python
import log1,json,logging.config;logging  = log1.get_clean_logging()
logging.config.dictConfig(json.load(open("conf_dict.json")))
log = logging.getLogger("")
child_A = logging.getLogger("A")
child_B = logging.getLogger("B")
child_B_A = logging.getLogger("B.A")
log.info("Now this is %s logging!","good")
child_A.info("Now this is more logging!")
log.warning("Now this is %s logging!","worrisome")
```

```
2015-07-19 20:19:55,865 INFO      root                   : Now this is good logging!
2015-07-19 20:19:55,866 INFO      A                      : Now this is more logging!
2015-07-19 20:19:55,867 WARNING   root                   : Now this is worrisome loggin
g!
```

Looking at the tree of Logging Objects



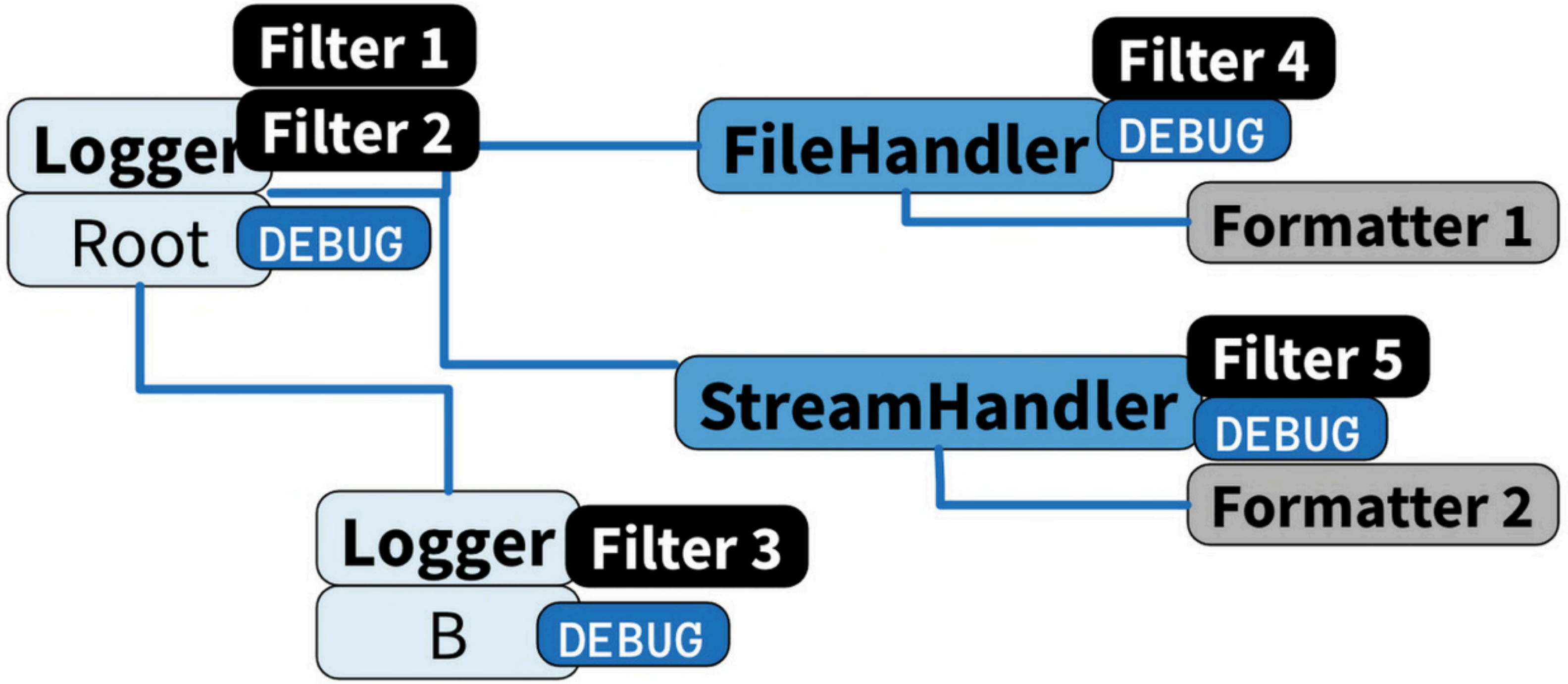
Best Practices for the Logging Tree

- Use `.getLogger(__name__)` per module to define loggers under the root logger
- Set propagate to True on each Logger
- Attach Handlers and Filters as needed to control output from the Logging hierarchy

Filter - Now that things are Getting Complicated

- With more loggers and handlers in the tree of logging objects, things are getting complicated
- We may not want every logger to send log records to every filter
- The logging level gives us some control, there are limits
- Filters are one solution to this problem
- Filter can also **add** information to records, thus helping with structured logging

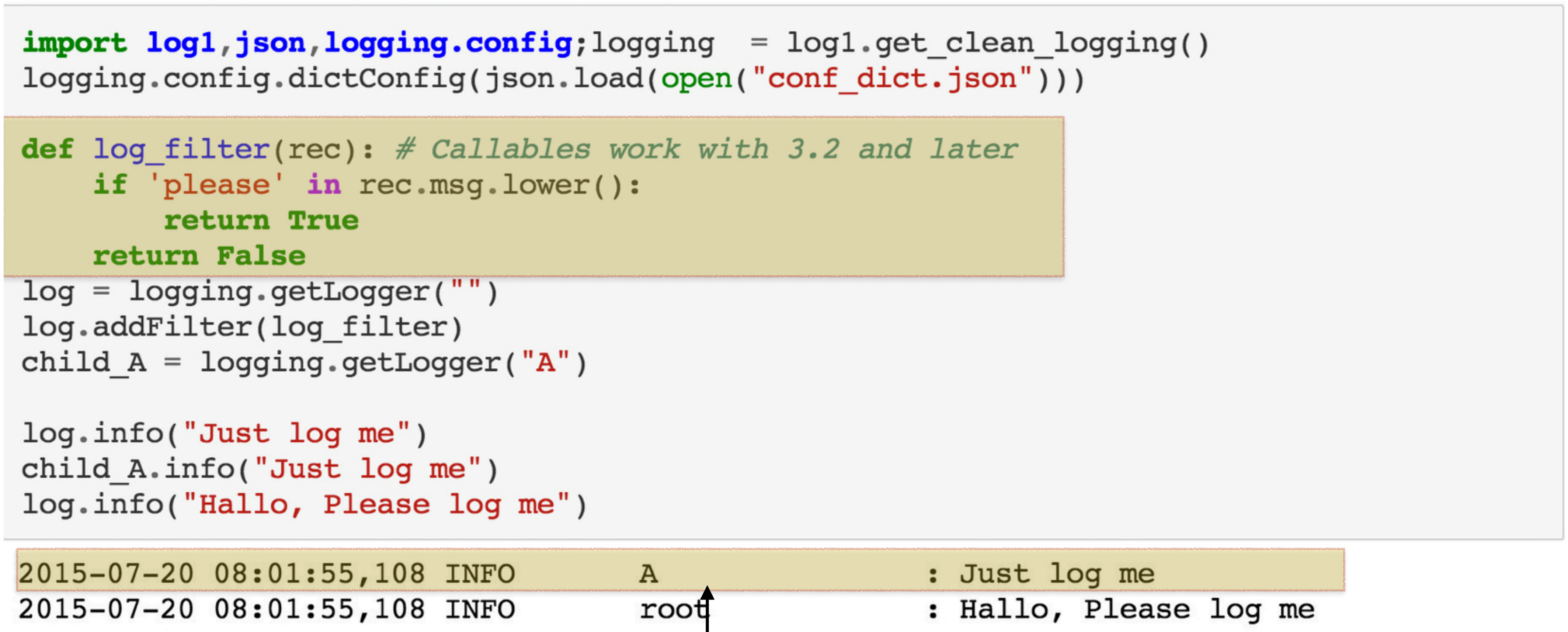
Using Filters



Filter = object with a `filter(record)` method.

Returns True or False.
Can modify in place to add / change/ remove information

An Example for using Filter Objects

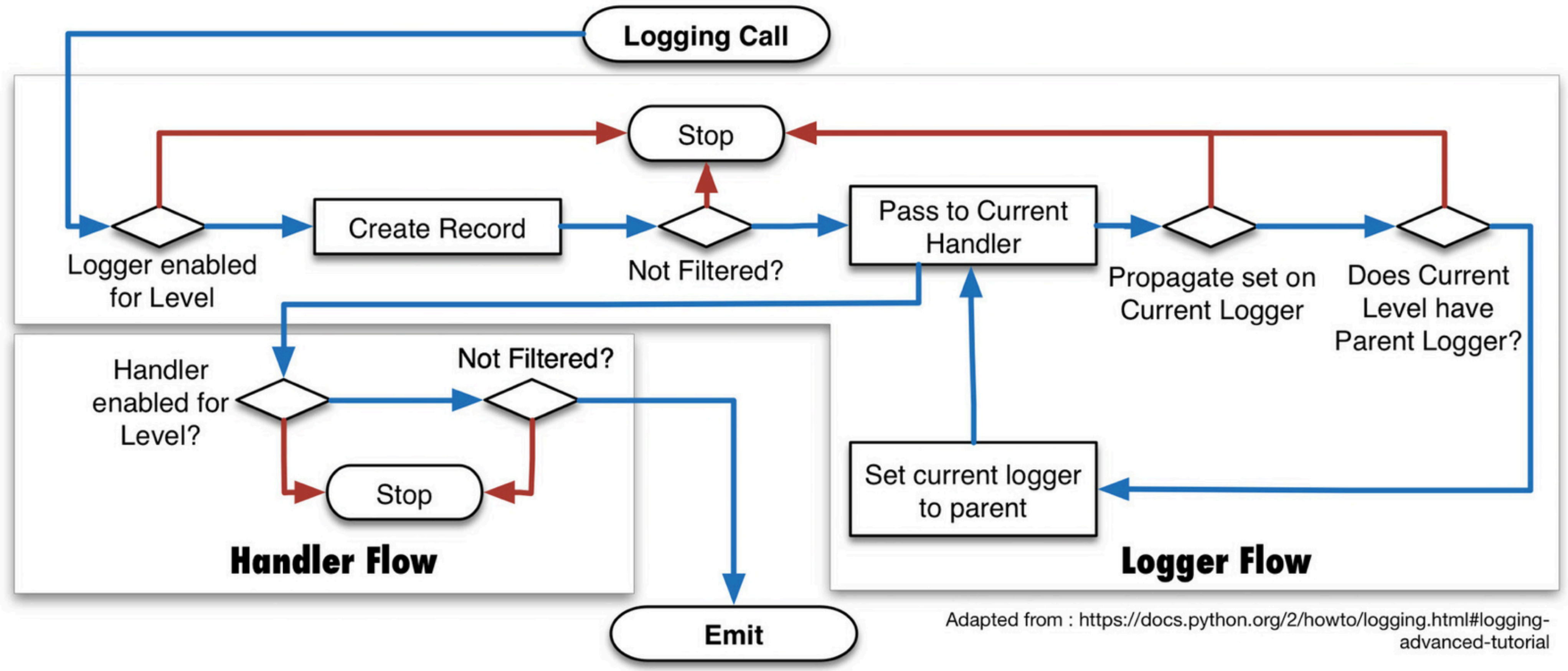


```python
import log1,json,logging.config;logging  = log1.get_clean_logging()
logging.config.dictConfig(json.load(open("conf_dict.json")))

def log_filter(rec): # Callables work with 3.2 and later
    if 'please' in rec.msg.lower():
        return True
    return False
log = logging.getLogger("")
log.addFilter(log_filter)
child_A = logging.getLogger("A")

log.info("Just log me")
child_A.info("Just log me")
log.info("Hallo, Please log me")
```

```
2015-07-20 08:01:55,108 INFO      A                : Just log me
2015-07-20 08:01:55,108 INFO      root             : Hallo, Please log me
```

Why?

The Way of a Logging Record



Adapted from : https://docs.python.org/2/howto/logging.html#logging-advanced-tutorial

A second Example for Filters, in the LogHandler

```python
import log1, json, logging.config;logging  = log1.get_clean_logging()
datefmt = "%Y-%m-%d %H:%M:%S"
msgfmt = "%(asctime)s,%(msecs)03d %(levelname)-6s %(name)-10s : %(message)s"
log_reg = None
def handler_filter(rec): # Callables work with 3.2 and later
    global log_reg
    if 'please' in rec.msg.lower():
        rec.msg = rec.msg + " (I am nice)" # Changing the record
        rec.args = (rec.args[0].upper(), rec.args[1] + 10)
        rec.__dict__['custom_name'] = "Important context information"
        log_reg  = rec
        return True
    return False
log = logging.getLogger()
lh = logging.StreamHandler()
lf =  logging.Formatter(fmt=msgfmt, datefmt=datefmt)
lh.setFormatter(lf)
log.addHandler(lh)
lh.addFilter(handler_filter)
log.warn("I am a bold Logger","good")
log.warn("Hi, I am %s. I am %i seconds old. Please log me","Loggy", 1)
```

```
2015-07-19 20:19:55,905 WARNING root       : Hi, I am LOGGY. I am 11 seconds old.
Please log me (I am nice)
```

A short look at our LogRecord

```python
print(log_reg)
log_reg.__dict__
```

```
<LogRecord: root, 30, <ipython-input-20-d1d101ab918f>, 25, "Hi, I am %s. I am %i
seconds old. Please log me (I am nice)">

 {'args': ('LOGGY', 11),
  'asctime': '2015-07-19 20:19:55',
  'created': 1437329995.905689,
  'custom_name': 'Important context information',
  'exc_info': None,
  'exc_text': None,
  'filename': '<ipython-input-20-d1d101ab918f>',
  'funcName': '<module>',
  'levelname': 'WARNING',
  'levelno': 30,
  'lineno': 25,
  'message': 'Hi, I am LOGGY. I am 11 seconds old. Please log me (I am nice)',
  'module': '<ipython-input-20-d1d101ab918f>',
  'msecs': 905.689001083374,
  'msg': 'Hi, I am %s. I am %i seconds old. Please log me (I am nice)',
  'name': 'root',
  'pathname': '<ipython-input-20-d1d101ab918f>',
  'process': 1644,
  'processName': 'MainProcess',
  'relativeCreated': 1.280069351196289,
  'stack_info': None,
  'thread': 140735243608832,
  'threadName': 'MainThread'}
```

Things you might want to know (if we still have some time)

Logging Performance - Slow, but Fast Enough

Scenario (10000 Call, 3 Logs per call)	Runtime
Full Logging with buffered writes	3.096s
Disable Caller information	2.868s
Check Logging Lvl before Call, Logging disabled	0.186s
Logging module level disabled	0.181s
No Logging calls at all	0.157s

Getting the current Logging Tree

```python
import json, logging.config
config = json.load(open("conf_dict_with_file.json"))
logging.config.dictConfig(config)
import requests
import logging_tree
logging_tree.printout()
```

```
<--""
   Level DEBUG
   Handler Stream <IPython.kernel.zmq.iostream.OutStream object at 0x105d043c8>
     Formatter fmt='%(asctime)s,%(msecs)03d %(levelname)-10s %(name)-15s : %(mess
age)s' datefmt='%Y-%m-%d %H:%M:%S'
   Handler File '/Users/imhiro/AllFiles/0021_travel_events_conferences_workshop
s/2015-07-19_europython/github/logfile.txt'
     Formatter fmt='%(asctime)s,%(msecs)03d %(levelname)-10s %(name)-15s : %(mess
age)s' datefmt='%Y-%m-%d %H:%M:%S'
   |
   o   "IPKernelApp"
   |   Level WARNING
   |   Propagate OFF
   |   Disabled
   |   Handler Stream <_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-
8'>
   |     Formatter <IPython.config.application.LevelFormatter object at 0x104b362
e8>
   |
   o<--[concurrent]
   |   |
```

Reconfiguration

- It is possible to change the logging configuration at runtime
- It is even part of the standard library
- Still, some caution is in order

Reloading the configuration *can* disable the existing loggers

```
import log1,json,logging,logging.config;logging  = log1.get_clean_logging()

#Load Config, define a child logger (could also be a module)
logging.config.dictConfig(json.load(open("conf_dict_with_file.json")))
child_log = logging.getLogger("somewhere")

#Reload Config
logging.config.dictConfig(json.load(open("conf_dict_with_file.json")))


#Our childlogger was disables
child_log.info("Now this is %s logging!","good")
```

Reloading can happen in place

```python
import log1, json, logging, logging.config;logging  = log1.get_clean_logging()

config = json.load(open("conf_dict_with_file.json"))
#Load Config, define a child logger (could also be a module)

logging.config.dictConfig(config)
child_log = logging.getLogger("somewhere")
config['disable_existing_loggers'] = False
#Reload Config
logging.config.dictConfig(config)



#Our childlogger was disables
child_log.info("Now this is %s logging!","good")
```

```
2015-07-19 20:20:42,290 INFO       somewhere       : Now this is good logging!
```

Successful Logging to all of You