Problem to Solve

Sujet2: Decorator The language to use is Python.

Écrire un décorateur nommé log_content , qui enregistre chaque variable de façon transparente à la fonction Exemple:

```
@log_content
def example(a, b, c):
    a = b + c
    b = a + c
    c = a + b
    d = "hi there"
    result = a + b + c
    return result

output:
[2018-03-16 14:37:00,666][file:test.py]DEBUG [@example] variable a = 5
[2018-03-16 14:37:00,975][file:test.py]DEBUG [@example] variable b = 4
[2018-03-16 14:37:00,978][file:test.py]DEBUG [@example] variable c = 3
[2018-03-16 14:37:00,978][file:test.py]DEBUG [@example] variable d = hi
there
[2018-03-16 14:37:00,978][file:test.py]DEBUG [@example] variable result =
12
```

Solution

We create our first class CustomFormatter, in which we define a get_logger() function. Here, we will create and return a logger object using theLogging module in python. In get_logger(), we will create a logger object using theLogger() function of theLogging module. And define our decorator in log_decorator() function .

using the customFormatter class

The logging decorator, logs the file and function name where it's defined not where it's used. For example, if we are using a decorator to log the file main.py and defined the log decorator in log_decorator.py. Then, in our log file even though we are logging main.py, it will take function name and file name of log_decorator.py from where we are calling. To get the actual file name and function name we have defined our own custom class named as customFormatter by overiding their values.

In [1]:

```python
import logging
import os

class CustomFormatter(logging.Formatter):

    def format(self, record):
        if hasattr(record, 'func_name_override'):
            record.funcName = record.func_name_override
        if hasattr(record, 'file_name_override'):
            record.filename = record.file_name_override
        return super(CustomFormatter, self).format(record)
```

After creating a logger object, we need to add a formatter and level in order to specify the format for our log output.

Add Formatter in get_logger():

In [2]:

```
def get_logger():

    # Create logger object and set the format for logging and other attributes
    logger = logging.Logger(__name__)
    handler = logging.StreamHandler()
    logger.setLevel(logging.DEBUG)
    handler.setLevel(logging.DEBUG)
    logging.getLogger().setLevel(logging.DEBUG)
    handler.setFormatter(CustomFormatter('[%(asctime)s][file:%(filename)s]%(levelna
    logger.addHandler(handler)

    # Return logger object
    return logger
```

Setting level:

There are various levels in python in which one can log details. When a logger is created, the level defaults to NOTSET.

LEVEL NUMERIC VALUE

CRITICAL : 50

ERROR : 40

WARNING : 30

INFO : 20

DEBUG : 10

NOTSET : 0

Only messages of or above the selected severity level will be emitted by whichever handler or handlers service the logger.

Let us set our level to DEBUG.

setting the level of logger

logger.setLevel(logging.DEBUG)

Inside the decorator, we will define a wrapper class in which we will call our function which we want to log.

We will create logger_obj by calling get_logger(). After that, we can simply log the function details using logger_obj. Here, we are logging start of execution the function, it’s return value.

Using extra argument in logger :

Now in order to get the correct function and file name, we will use the'extra',in-built argument of the logger object. Using this parameter, we can get the used function and file names.

In [3]:

```
import sys, os, functools
from inspect import getframeinfo, stack

def log_decorator(_func=None):
    def log_decorator_info(func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            # Build logger object
            logger_obj = get_logger()

            result = 0
            py_file_caller = getframeinfo(stack()[1][0])
            extra_args = { 'func_name_override': func.__name__,
                           'file_name_override': os.path.basename(py_file_caller.filena
            for key, value in kwargs.items():
                logger_obj.debug("variable {} = {}" .format (key,value), extra=extra_ar
                result = value+result
            value = func(*args, **kwargs)
            logger_obj.debug(f"variable d = hi there", extra=extra_args)
            logger_obj.debug(f"variable result = {result!r}", extra=extra_args)
        # Return the pointer to the function
        return wrapper
    # Decorator was called with arguments, so return a decorator function that can read
    if _func is None:
        return log_decorator_info
    # Decorator was called without arguments, so apply the decorator to the function im
    else:
        return log_decorator_info(_func)
```

We are getting file name using getframeinfo() routine of module inspect and function name using **func**.name and store it in extra_args variable. In logger_obj we will pass extra_args as an extra arg, which will override the names using customFormatter class.

In [4]:

```
class Log():
    @log_decorator
    def example(a,b,c):
        a = b + c
        b = a + c
        c = a + b
        d = "hi there"
        result = a + b + c
        return result

    example(a=5,b=4,c=3)
```

```
[2021-01-03 13:12:19,350][file:<ipython-input-4-40ecf87b2959>]DEBUG [@example] variable
a = 5
[2021-01-03 13:12:19,353][file:<ipython-input-4-40ecf87b2959>]DEBUG [@example] variable
b = 4
[2021-01-03 13:12:19,354][file:<ipython-input-4-40ecf87b2959>]DEBUG [@example] variable
c = 3
[2021-01-03 13:12:19,355][file:<ipython-input-4-40ecf87b2959>]DEBUG [@example] variable
d = hi there
[2021-01-03 13:12:19,357][file:<ipython-input-4-40ecf87b2959>]DEBUG [@example] variable
result = 12
```

NB: The file name doesn't display properly because of Jupyter Notebook