Mittuniversitetet

MID SWEDEN UNIVERSITY

DT179G, Programming Fundamentals

Laboration 3

The objectives of the laboration aim to provide practical experience in regards to…

- … iterative and recursive implementations!
- … working with function decoration!
- … file handling!
- … usage of custom loggers!
- … separation of concerns!
- … creation of well structured program code!

Table of Contents

Laboration 3....3
Background....4
Fibonacci Sequence Measurements....5
Assignment: Fibonacci Sequences....6
Inherent Code....7
Fibonacci Functions....8
Create Logger....9
Measurements Decorator....10
Printing the Stats....11
Writing Data to Files....12
Report Purpose....13
English Version....13
Swedish Version....13
Requirements....14
Submission....15

Laboration 3

> *Before starting to work on the laboration you should have studied the module's belonging lecture material and read relevant chapters in the course literature, as well as completed the corresponding exercises.*

The lab session comprises a single assignment, and you'll be required to present the solution in the specified file within your student repository. Pertinent files for this lab include:

`_Resources/ass3_log.conf.json`

`Laboration_3/assignment.py`

`Laboration_3/README.md`

Adherence to the proper file structure is mandatory, and you must not alter folder or file names. Ensure that your code is placed within the specified source file. Additionally, the **`README`** needs to detail your implementation approach and offer both analyses and reflections on the assignment. You can find more specifics on the report structure in the study guidance.

Since Python is a dynamic language we don't need to explicitly state datatypes, but having some indication of expected output will indeed clarify intended behavior and overall purpose. The type hint `-> bool` tells us that a boolean value should be returned to the calling client. You may regard this as a kind of contract, where the function promise to provide a certain output. However, Python does not enforce these kinds of type checks so you may break such contracts freely without any real repercussions.

In this course we'll utilize these type hints to indicate which datatypes each entity needs to return, and by that clarify the requirements. Declaring return types also has another benefit where statical analyses, such as PyLint, can use them to determine specification conformities. If you're using PyCharm, the IDE will also notify whenever expected data type is not returned!

Background

The Fibonacci sequence has a wide spectrum of applications throughout different scientific disciplines. Because of its conformity to the ***golden ratio*** [1] it can be used to describe various natural patterns such as branching of trees in biology, molecular compositions in chemistry, and the formation of galaxies in astrophysics. It's even used in computer science for different search techniques and to represent efficient data structures.

The Fibonacci Sequence is the series of numbers...

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

… where the next number in the sequence is determined by adding up the two numbers before it. The next number in above sequence should therefore be **21 + 34 = 55**. This rule can be described as…

$$X_n = X_{n-1} + X_{n-2}$$

… where **n** is a representation of the sequential order, which gives us the following mapping:

X_0	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	X_{10}	X_{11}	X_{12}	X_{13}	X_{14}	...
0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	...

In order of determining the **n**'th consecutive Fibonacci number there are different methods we could apply, such as **iterative** and **recursive** approaches. You've already worked with iteration when implementing loops during the exercises and recursion is very similar to this in concept, but involves a function to call upon itself until a certain condition is met. Recursion can actually be used as an alternative to iteration in most cases and the following two snippets show how factorials can be calculated using both approaches:

```python
def factorial_iteration(n):
    fact = 1
    for i in range(1, n + 1):
        fact = fact * i
    return fact
```

```python
def factorial_recursion(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial_recursion(n - 1)
```

There are however some subtle, but still important differences between the two concepts. Repeated recursion implies an overhead of method calls, which could become exponentially more expensive both in terms of processing time and memory space since it involves stack allocation for each call. On the other hand, recursion can help keep the code base smaller and more maintainable by factoring out responsibilities to dedicated (and named) entities. Recursion can also be quite valuable in terms of algorithmic design, since it may alleviate some of the complexities usually associated with deeply nested loops.

It's always important to find a balance between performance and maintainability, while also striving to minimize complexity. Use the right tool for each challenge and while we won't dive into code optimization in this course, it could be interesting to study the efficiencies between different algorithms. In this laboration we will measure execution time between distinct implementations of the same procedure, just to get a feel for the differences.

1 Golden Ratio, Widipedia, https://en.wikipedia.org/wiki/Golden_ratio

Fibonacci Sequence Measurements

In order to measure execution time we need to use timestamps to mark both the start and end of the procedures. The duration can then be determined simply by subtracting the start time from end time. For these purposes we'll use **`default_timer`** from module **`timeit`**, which will provide a timer object specific for the system you're using.

```
from timeit import default_timer as timer
```

The functions **`fibonacci_iterative`**`(nth_nmb)` and **`fibonacci_recursive`**`(nth_nmb)`, which you see below, constitute two different approaches to calculate Fibonacci sequences, much like the factorial examples from above. The parameter **`nth_nmb`** represents the max number to sequence and both functions run calculations for sequences within the interval **`0 – nth_nmb`** before returning requested value to the calling client.

```
def fibonacci_iterative(nth_nmb):
    old, new = 0, 1
    if nth_nmb == 0 in nth_nmb == 1:
        return nth_nmb
    for nmb in range(nth_nmb):
        old, new = new, old + new
    return new
```

```
def fibonacci_recursive(nth_nmb):
    if nth_nmb <= 1:
        return nth_nmb
    else:
        return fibonacci_recursive(nth_nmb - 1) + \
               fibonacci_recursive(nth_nmb - 2)
```

In order to time each execution we could use a function devoted for measurements, seen below with example of usage. The return value will be a **`timer`** object representing the duration in seconds.

```
def run_measurements(fibonacci, nth_nmb):  # accepts function reference as param
    start_time = timer()            # mark start time
    for nmb in range(nth_nmb + 1):  # iterate the interval 0 - nth_nmb
        fibonacci(nmb)              # calc sequence for number
    return timer() - start_time     # return duration (float)


nth_value = 30

results = {  # store result for each approach in dictionary
    'iteration': run_measurements(fibonacci_iterative, nth_value),
    'recursion': run_measurements(fibonacci_recursive, nth_value)
}

# Print type of approach and duration (limited to 5 decimals) for each execution
for approach, duration in results.items():
    print("{}: {:.5f} seconds".format(approach.title(), duration))
```

Example output

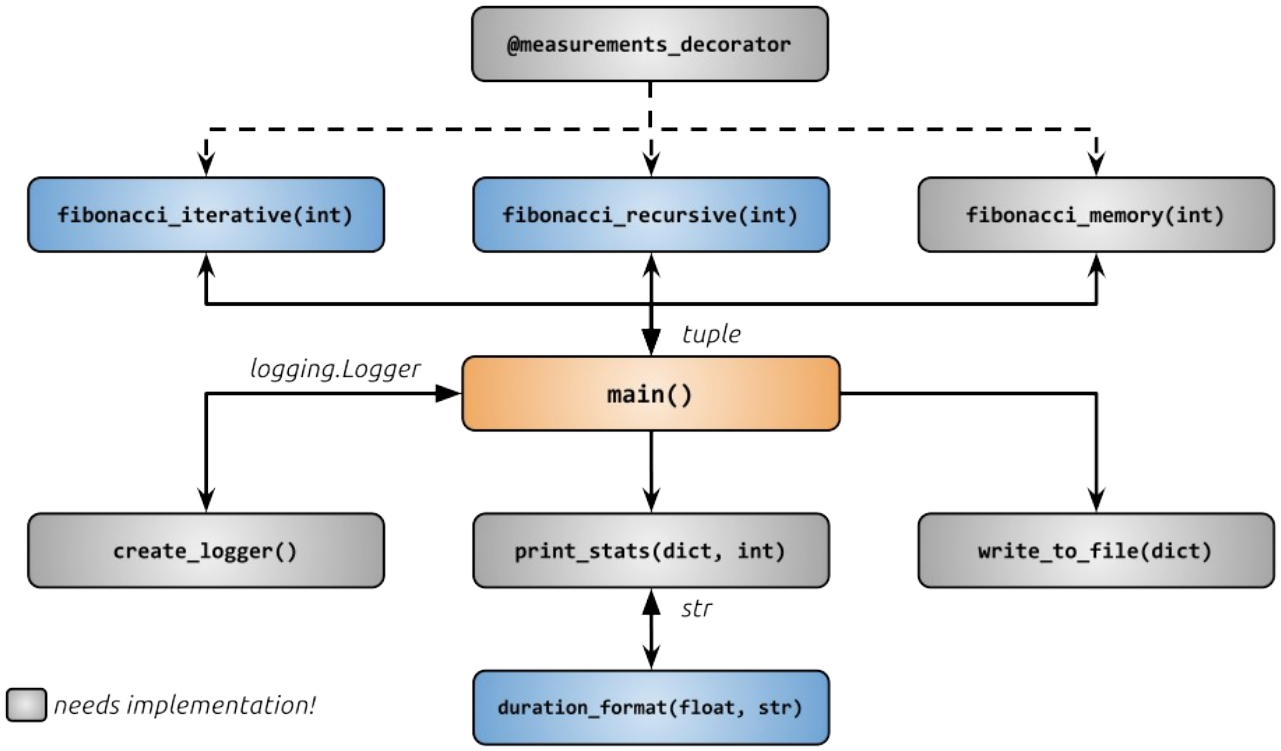
```
Iteration: 0.00003 seconds
Recursion: 0.81870 seconds
```

As you can see the iterative approach is way faster than the recursive, taking **`~0.004%`** of the time to run full execution. In terms of microseconds (**`µs`**: millionth of a second) this translates to **`30µs`** for the iterative, while the recursive requires a staggering **`818700µs`** to complete. For the assignment you will create one additional approach, as a middle ground, and implement the measurements by decorating the functions.

Assignment: Fibonacci Sequences

For the assignment you need to create a program which can measure the execution time of Fibonacci sequence calculations using three different approaches. The iterative and recursive approaches from previous chapter is already defined, though implemented in slightly different manners, and you may not modify these in any way. However, you will need to complement these approaches with an additional variation of recursion which keeps a memory of calculated sequences, thus speeding up execution time.

The behaviors of these three approaches also needs to be extended using a single decorator, which will run time measurements and communicate with a custom logger. The diagram below describes all functions needed for the assignment, and you need to implement those marked with gray color in conformity to requirements stated throughout this document.



You can start the program either within the IDE, such as PyCharm, or from the command line:

```
python assignment.py      # run program using default nth value (30)
python assignment.py 20   # run program, stating nth value
python assignment.py -h   # show help screen
```

The expected output from running the application can be seen under Requirements!

Inherent Code

```
RESOURCES = Path(__file__).parent / "../_Resources/"
if __name__ == "__main__":
    main()
def main():
def duration_format(duration: float, precision: str) -> str:
```

Above entities are already defined in **Laboration_3/assignment.py** and you may not alter or modify their specifications or implementations in any way. The variable **RESOURCES** will have module scope and simply states the path to resource folder. Besides these entities, there are the needed imports for the assignment and some other functions which will be described in later sections of this document.

```
def main():
    """ The main program execution. YOU MAY NOT MODIFY ANYTHING IN THIS FUNCTION!! """
    ... # code for argparse. For details, study assignment.py
    global LOGGER  # ignore warnings raised from linters, such as PyLint!
    LOGGER = create_logger()

    args = parser.parse_args()
    nth_value = args.nth  # nth value to sequence. Will fallback on default value!

    fib_details = {  # store measurement information in a dictionary
        'fib iteration': fibonacci_iterative(nth_value),
        'fib recursion': fibonacci_recursive(nth_value),
        'fib memory': fibonacci_memory(nth_value)
    }

    print_statistics(fib_details, nth_value)    # print information in console
    write_to_file(fib_details)                  # write data files
```

The function **main()** will handle the main execution, such as parsing command arguments and calling needed functions. Of note here is that the **LOGGER** is made **global**, which makes it accessible to all entities. **nth_value** refers to the Fibonacci sequence we are aiming for, and will be parsed from command argument (defaulting to **30** if none is provided). The results of running all three Fibonacci approaches will be stored in a dictionary, and later sent to procedures for both printing statistics on screen and writing information to files. Below, you also see the definition of **duration_format(..)** which will be used to convert numeric values into formatted strings.

```
def duration_format(duration: float, precision: str) -> str:
    """ Function to convert number into string. Switcher is dictionary type here.
    YOU MAY NOT MODIFY ANYTHING IN THIS FUNCTION!! """
    switcher = {
        'Seconds': "{:.5f}".format(duration),
        'Milliseconds': "{:.5f}".format(duration * 1_000),
        'Microseconds': "{:.1f}".format(duration * 1_000_000),
        'Nanoseconds': "{:d}".format(int(duration * 1_000_000_000))
    }
    return switcher.get(precision, "nothing")
```

Fibonacci Functions

```
def fibonacci_iterative(nth_nmb: int) -> int:
def fibonacci_recursive(nth_nmb: int) -> int:
def fibonacci_memory(nth_nmb: int) -> int:
```

Arg	Return
...	...
6	8
7	13
8	21
...	...

```
Input to each function will be the n'th consecutive Fibonacci sequence to
find, while the output will be the result from this search. We're only
interested in finding positive values.
```

The functions **`fibonacci_iterative(..)`** and **`fibonacci_recursive(..)`** represent two of the approaches to be used for calculating sequences, and they are both fully defined in your student repo. The provided implementation for both of these procedures should not be altered in any way, but you need to understand them in order to implement the third approach.

```python
@measurements_decorator
def fibonacci_iterative(nth_nmb: int) -> int:
    old, new = 0, 1
    if nth_nmb in (0, 1):
        return nth_nmb
    for __ in range(nth_nmb - 1):
        old, new = new, old + new
    return new
```

```python
@measurements_decorator
def fibonacci_recursive(nth_nmb: int) -> int:
    def fib(_n):
        return _n if _n <= 1 else \
            fib(_n - 1) + fib(_n - 2)
    return fib(nth_nmb)
```

As you will see from closer inspection both of these implementations conforms to respective example from earlier chapter, though with some slight differences. The most notable difference is that the recursive implementation relies on an inner function. The functions are also decorated, but we'll worry about that in later section. For now, lets focus on the third Fibonacci approach which lacks implementation.

```python
@measurements_decorator
def fibonacci_memory(nth_nmb: int) -> int:
    """A recursive approach to find Fibonacci sequence value, storing those already calculated."""
    pass  # TODO: Replace with implementation!
```

You need to write the full logic for **`fibonacci_memory(..)`**, but may not modify its specification (*name*, *parameters* and *return type*). Just like the other two approaches, the function accepts one argument of integer type and use the same data type as return value. Your challenge here is to make a more efficient algorithm of the recursive implementation, by storing / caching already calculated values for easy access in later iterations. Note that the logic should be nearly identical to above recursive implementation, with the only difference that sequences together with calculated values are stored in a container!

Pseudo Code

```
* DEFINE container for values
  IF NOT n'th value EXIST in container:
    CALCULATE value AND STORE in container
  RETURN n'th value
```

* Use a dictionary to store sequences and calculated values in a dictionary. In order to avoid **`RecursionError`** it may also be appropriate to initialize the container with values for **`0`** and **`1`**, but note that no other values may be hard-coded:

```python
memory = {0: 0, 1: 1}
```

Create Logger

```python
def create_logger() -> logging.Logger:
    """ Create and return logger object. """
    pass  # TODO: Replace with implementation!
```

We will use a custom logger to output information throughout running the measurements. Name of logger needs to be **`ass_3_logger`** and it needs to be configured with two handlers, one for console output (**`INFO`** level) and the other for file output (**`DEBUG`** level). You may create the logger, with handlers, either by manual configuration or by using the existing configuration file **`_Resources/ass3_log_conf.json`**. If you configure the logger manually you also need to **`import sys`**, since the console handler will pipe information directly to **`sys.stdout`**. The file handler needs to output information to **`_Resources/ass_3.log`**.

Contents of _Resources/ass_3.log running sequences 20..0

```
2020-11-25 15:52:19 | ass_3_logger | INFO  | Starting measurements...
2020-11-25 15:52:19 | ass_3_logger | DEBUG | 20: 6765
2020-11-25 15:52:19 | ass_3_logger | DEBUG | 15: 610
2020-11-25 15:52:19 | ass_3_logger | DEBUG | 10: 55
2020-11-25 15:52:19 | ass_3_logger | DEBUG | 5: 5
2020-11-25 15:52:19 | ass_3_logger | DEBUG | 0: 0
2020-11-25 15:52:19 | ass_3_logger | INFO  | Starting measurements...
2020-11-25 15:52:20 | ass_3_logger | DEBUG | 20: 6765
2020-11-25 15:52:20 | ass_3_logger | DEBUG | 15: 610
2020-11-25 15:52:20 | ass_3_logger | DEBUG | 10: 55
2020-11-25 15:52:20 | ass_3_logger | DEBUG | 5: 5
2020-11-25 15:52:20 | ass_3_logger | DEBUG | 0: 0
2020-11-25 15:52:20 | ass_3_logger | INFO  | Starting measurements...
2020-11-25 15:52:20 | ass_3_logger | DEBUG | 20: 6765
2020-11-25 15:52:20 | ass_3_logger | DEBUG | 15: 610
2020-11-25 15:52:20 | ass_3_logger | DEBUG | 10: 55
2020-11-25 15:52:20 | ass_3_logger | DEBUG | 5: 5
2020-11-25 15:52:20 | ass_3_logger | DEBUG | 0: 0
```

The colors shown in this example is used to make the separation of each part more understandable, and since the file handler will be configured to **`DEBUG`** level it will also pick up logs for higher levels. Each log in the file will consist of the following parts, separated by a vertical line which is both preceded and exceeded by an empty space (**`“ | “`**):

`timestamp | logger_name | log_level | message`

`timestamp`	needs to conform to format **`“%Y-%m-%d %H:%M:%S”`**. Note the space between date and time!
`logger_name`	needs to be **`ass_3_logger`** and nothing else. The name needs to be pulled from the logger configuration!
`log_level`	refers to log severity, either **`INFO`** or **`DEBUG`**, and will be determined by calling client!
`message`	the message to output, will be provided from calling client!

The formatter for file handler needs to use all of these parts, while the console formatter will only use **`log_level`** and **`message`**.

If logger is created from config file you may want to change relative path for output file for **`file_handler`** to an absolute path instead. If you’re running the program from terminal, this will enable execution no matter of current working directory!

Measurements Decorator

```python
def measurements_decorator(func):
    """ Function decorator, used for time measurements. """
    @wraps(func)
    def wrapper(nth_nmb: int) -> tuple:
        pass  # TODO: Replace with implementation!

    return wrapper
```

Each Fibonacci function is decorated with **`measurements_decorator(..)`**, which will be responsible for measuring execution time and trigger the logging of information. You will need to write an implementation to the i**nner wrapper,** in accordance to below pseudo code, and make sure correct information is sent to the logger.

Pseudo Code for wrapper

```
DEFINE list container for values
MARK starttime
LOG “Starting measurements...” at INFO level
* REPEAT IN RANGE nth_nmb - 0:
    DETERMINE fibonacci value AND add to container
**  FOR EACH 5th iteration LOG information at DEBUG level
MARK endtime AND calculate duration
RETURN duration AND container AS tuple
```

* Repeat execution from **`nth_nmb`** until (and including) **`0`**. If **`nth_nmb`** to find is **`30`**, then all Fibonacci values within the sequence range of **`30..0`** will be included. In other words, the iteration needs to be performed in reversed order!

** The information sent to the logger needs to contain Fibonacci sequence and its determined value, in accordance to the following format (note the space): **`seq: val`**

Example:

```
20: 6765
15: 610
10: 55
5: 5
0: 0
```

The time spent executing the full range of sequences (**`nth_nmb..0`**) for the approach in question will be measured and its duration calculated. The value of each Fibonacci sequence within the interval will be stored in a list. The final act of this **wrapper** is to return executions duration and list of values as a tuple to calling client.

> This part of the assignment will prove quite challenging unless you have paid close attention to both lecture material and examples belonging to Module 6. Whenever you feel lost or stuck, return to these resources and study them with a new perspective!

Printing the Stats

```
def print_statistics(fib_details: dict, nth_value: int):
    """ Function which handles printing to console. """
    line = '\n' + ("---------------" * 5)
    pass  # TODO: Replace with implementation!
```

Param	Description
`fib_details: dict`	Measurement details in the form: `{"approach": (duration: float, values: list)}`
`nth_value: int`	The max nth Fibonacci value to sequence!

```
--------------------------------------------------------------------------------
                DURATION FOR EACH APPROACH WITHIN INTERVAL: 30-0
--------------------------------------------------------------------------------
                    Seconds     Milliseconds      Microseconds       Nanoseconds
Fib Iteration       0.00018          0.18485             184.9            184852
Fib Recursion       0.74276        742.76093          742760.9         742760927
Fib Memory          0.00038          0.37702             377.0            377021
```

`print_statistics(..)` will be responsible for printing stats information from running the measurements to the screen, as shown above. The colors are simply used to emphasize different parts of the output, but take note of how the information is formatted and justified:

Type	Formatting Rules
Main header	In all upper case letters and with centered justification, and the sequence interval used for measurements. **`line`**, as it's defined, will be used for the lines above and below the text.
Column headers	The precisions used for various duration formats: **`Seconds`**, **`Milliseconds`**, **`Microseconds`**, **`Nanoseconds`**. All justified to the right and beginning with an upper case letter.
Row headers	Will be the **`"approach"`** from parameter **`fib_details`**. Needs to be left justified and formatted so that first letter of both words is upper case.
Duration values	Right justified and retrieved from **`duration_format(..)`**, using relevant precision and the base duration as arguments.

In order to determine width of justification for right aligned information, it may be appropriate to base it on the column header with most characters. Also remember that attributes can be stated for **`str.format()`**:

```
"{:>{width}}".format(width=col_width)
```

Writing Data to Files

```python
def write_to_file(fib_details: dict):
    """ Function to write information to file. """
    pass  # TODO: Replace with implementation!
```

The function **`write_to_file(..)`** accepts Fibonacci details as argument and will create corresponding files containing calculated values. Three files shall be created, one for each approach, and their names will consist of a composition between the keys of **`fib_details`** and **`.txt`**. All files must be saved in the **`_Resources`** directory of your student repository, and no space is allowed in the filename (replace with underscore):

- **`_Resources/fib_iteration.txt`**
- **`_Resources/fib_memory.txt`**
- **`_Resources/fib_recursion.txt`**

All files must have identical contents and the information needs to be stored in conformity with the format shown to the right: **`sequence: value`**. ***Note the empty space!***

```
30: 832040
29: 514229
28: 317811
27: 196418
26: 121393
25: 75025
24: 46368
23: 28657
22: 17711
21: 10946
20: 6765
19: 4181
18: 2584
17: 1597
16: 987
15: 610
14: 377
13: 233
12: 144
11: 89
10: 55
9: 34
8: 21
7: 13
6: 8
5: 5
4: 3
3: 2
2: 1
1: 1
0: 0
```

Pseudo Code

```
FOR EACH Fibonacci approach:
  WITH stream to file:
    FOR sequence AND value:
      WRITE sequence and value to file
```

One challenge here is to match each sequence number to its corresponding value. Since the function only accepts one argument, the amount of sequences needs to be retrieved from **`fib_details`**. But simply traversing the value list contents will only get the correct values, and to map it to the corresponding sequence number we need to adjust the approach.

You could of course use dual counters for the iteration, where one is incremented and the other decremented. But another function which is particularly useful for these matters is **`zip()`**, which you find some examples of in your exercise repo!

Report Purpose

English Version

This lab explores the distinctions between iterative and recursive methods, the intricacies of function decoration, efficient file handling, and the practical use of custom loggers. Centered on various Fibonacci sequencing techniques, the lab's aim is to accurately measure execution times and present pertinent data through the console and log files. The overarching objective is to design and craft a system that consistently measures, logs, and showcases this data to the user. The concrete goals for determining success of this study are:

- Initialize and configure `ass_3_logger` with its dual handlers. Ensure the console outputs at the `INFO` level and file outputs at the `DEBUG` level, adhering to the defined formatting standards like timestamp, logger name, log level, and message content.
- Implement the `fibonacci_memory(nth_nmb)` function as per the requirements. This function should offer a performance edge over the standard recursive method by utilizing memory, leading to reduced execution time.
- Implement the `@measurements_decorator` to monitor the execution times of each Fibonacci function. This decorator needs to align with the provided pseudo code, log appropriate information, capture Fibonacci sequences, compute durations, and return the correct data to the caller.
- Implement the `print_statistics(fib_details: dict, nth_nmb: int)` function, adhering to the stipulated guidelines. This function needs to display sequencing statistics in the console, using the designated formatting and the supplied helper function for time formatting nuances.
- Implement the `write_to_file(fib_details: dict)` function as per the requirements. This function needs to generate files with exact Fibonacci details, conform to the specified naming conventions, and store them in the `_Resources` directory, all while ensuring content format consistency.

Swedish Version

Denna laboratorion utforskar skillnaderna mellan iterativa och rekursiva metoder, nyanser kring funktiondekorering, effektiv filhantering och den praktiska användningen av anpassade loggers. Med fokus på olika Fibonacci-sekvenseringstekniker är arbetets övergripande mål att exakt mäta utförandetider och presentera relevant data genom såväl konsol som loggfiler. Det övergripande målet är att designa och skapa ett system som konsekvent mäter, loggar och visar denna data för användaren. De konkreta målen för att avgöra studiens framgång är:

- Initiera och konfigurera `ass_3_logger` med dess dubbla hanterare. Se till att konsolen ger utdata på `INFO`-nivå och filutdata på `DEBUG`-nivå, samt håller sig till de definierade formatstandarderna som tidsstämpel, loggernamn, loggnivå och meddelandeinnehåll.
- Implementera funktionen `fibonacci_memory(nth_nmb)` enligt kraven. Denna funktion skall erbjuda en prestandafördel jämfört med den standard rekursiva metoden genom att använda minne, vilket leder till kortare utförandetid.
- Implementera `@measurements_decorator` för att övervaka utförandetiderna för varje Fibonacci-funktion. Denna dekoratör behöver stämma överens med den tillhandahållna pseudokoden, logga lämplig information, fånga Fibonacci-sekvenser, beräkna varaktigheter och returnera rätt data till anroparen.
- Implementera funktionen `print_statistics(fib_details: dict, nth_nmb: int)`, i enlighet med de angivna riktlinjerna. Denna funktion behöver visa sekvenseringsstatistik i konsolen, använda det angivna formatet och den tillhandahållna hjälpfunktionen för tidformateringsnyanser.
- Implementera funktionen `write_to_file(fib_details: dict)` enligt kraven. Denna funktion behöver generera filer med exakta Fibonacci-detaljer, följa de angivna namnkonventionerna och lagra dem i `_Resources`-katalogen, med konsekvent säkerställd innehållsformatering.

Requirements

All implementations needs to fulfill the requirements as described within devoted sections of this document. Besides this, there are more general requirements as well:

- You ***may not*** change or modify names of inherent files and folders within your student repository!
- You ***may not*** change or modify existing code, only add to stated entities. This restriction also includes specifications (*name*, *parameters*, *return value*) of all inherent functions!
- You may create additional functions as deemed necessary, but you ***may not*** circumvent the intended relationship between inherent entities. This means that you may factor out parts of a function's responsibility to other functions, as long as the specification (*parameters*, *return value*) of original function is retained!
- You may use as many variables as you wish for the assignment, but they may only be passed between entities as arguments and return values. ***No additional global / module scoped variables will be allowed!***
- `timeit.default_timer` needs to be used to measure time spent for executions!
- In terms of scalability, you may assume only positive Fibonacci sequences will be calculated!
- You need to add comments to the implementations you write, describing its logic and the author's intentions!
- Inherent `pass` statements and TODO comments in the source file needs to be replaced by suitable implementations. No `pass` statements or TODO's are to remain upon submission.
- `Laboration_3/README.md` needs to conform to requirements stated in the study guidance. This refers to both structure and contents. You need to use the pre-defined purpose provided in this document. Select either English or Swedish, depending on which language is relevant, and you may copy the content directly into your report, under heading Purpose. Just make sure Markdown syntax is maintained, such as proper syntax for bullet points.
- The solution, in its entirety, needs to reside within `master` branch of your repository. You may of course, and are encouraged to, conduct work in dedicated development branches, but need to merge this with `master` upon submission. Evaluation will only be performed on `master` branch!

The console output from running the application should produce the following (durations may vary between runs):

```
INFO | Starting measurements...
INFO | Starting measurements...
INFO | Starting measurements...
--------------------------------------------------------------------------------
                  DURATION FOR EACH APPROACH WITHIN INTERVAL: 30-0
--------------------------------------------------------------------------------
                      Seconds       Milliseconds       Microseconds        Nanoseconds
Fib Iteration         0.00018            0.18485              184.9             184852
Fib Recursion         0.74276          742.76093           742760.9          742760927
Fib Memory            0.00038            0.37702              377.0             377021
```

Submission

All local changes should continuously be committed with suitable messages and it's important that these local changes are synchronized with the repository's ***remote origin*** before submitting the solution for evaluation. When ready, you need to formally submit the work in Moodle by using the dedicated submission box. You should not attach any files to this formal submission, as the solution's state will be pulled directly from your student repository.

The program code needs to be well structured and have a layout in accordance to proper conventions. This concerns *tabbing and white spaces*, *describing name conventions*, *extra line-breaks* and *suitable comments* to support readability!

All work needs to be conducted individually by each student, but you are encouraged to both discuss and support each other using various platforms for communication!