Sequential execution

In the the general case, Python instructions are ran **one after another**. If an instruction 'A' follows a 'B' instruction, 'B' must wait for 'A' to finish in order to start. For instance:

In [1]:
```
from datetime import datetime
from time import sleep

def A():
    current_time = datetime.now().strftime('%H:%M:%S')
    print(f"[{current_time}] I am 'A', and I will sleep for 3 seconds")
    sleep(3)
    current_time = datetime.now().strftime('%H:%M:%S')
    print(f"[{current_time}] I am 'A', and my sleeping time was so good!")


def B():
    current_time = datetime.now().strftime('%H:%M:%S')
    print(f"[{current_time}] I am 'B'")

A()
B()
```

```
[20:08:54] I am 'A', and I will sleep for 3 seconds

[20:08:57] I am 'A', and my sleeping time was so good!
[20:08:57] I am 'B'
```

Explanation: 'B' must wait for the end of 'A' to start.

But 'A' is mainly useless! In fact, 'A' is doing something that **does not require the CPU** (`sleep`). Yet, **'A' won't let 'B' use the CPU** while it is sleeping.

Other execution modes

Threading

In the example above, it would be really cool to use the CPU while 'A' does not need it, though 'A' is running.

This is one side of what is called **concurrency**: several instructions are 'racing' to get some CPU computations as soon as they finished their non-CPU activity. In Python, the `threading` package can be used to write concurrent code.

A `thread` is some tasks that make sense to group together. One can start several threads that will access the CPU whenever it is available without having to wait for the other thread to be terminated.

This execution mode **provides speed up capabilities mainly for IO bound problems**. Indeed, while a thread is using the disk or waiting for the network, another one can use the CPU.

Multi-processing

Sometimes one may want some instructions to be ran in a completely independant way. The idea, in the example above, is that it would not be a problem for 'A' to occupy the CPU since another part of the CPU could be used to process 'B' at the same time. This is called **parallelism**. These 'parts' are the CPU cores. In Python, this is done using the `multiprocessing` package.

This execution mode can speed up several CPU intensive problems, i.e. **CPU bound**.

Limitations

Memory

`threading` and `multiprocessing` are interesting but may lead to memory errors:

- using `threading`, 2 threads might need to access almost at the same time the same variable, which can lead to data corruption.
- using `multiprocessing`, 2 processes might want to share some memory content, but this is not a native behaviour since a process is not allowed to access the memory content of another one.

Some functions were defined to share variables at no risk. But using them can be difficult. Moreover, the risk of making a mistake is rather high and may lead to inacurrate scientific results.

For this reason, **concurrency and parallelism must be the last resort methods to speed up a code**.

Overhead

Overhead running time associated with `multiprocessing` are **very large, especially for codes dealing with very large amount of data**. Consequently, multiprocessing is really useful for CPU-bound tasks that typically takes a few minutes to run. Conversely, threading comes with a lower overhead.

Advanced note

Python is a particular case regarding parallelism/concurrency. Indeed, the true limitation of Python is that a single Python process cannot run different instructions simultaneously on the CPU, due to an internal limitation. This explains the limitations of `threading` and the need for `multiprocessing`.

In other languages, threads can run simultaneously on different cores of the CPU. Then the difference between threads and processes is more related to memory management.