`VerboseManager`: a class for managing progress output on complex processes

A. H. Room

December 2021

1 Introduction

`VerboseManager` is a class made for managing verbose output on a complex Python method or function (i.e. one which goes into sub-functions). It allows a process to 'carry' its own verbose management through these sub-functions.

2 Basic process management

For a simple process with no sub-functions (or at least, none that would require their own verbose handling), `VerboseManager` handles management like so:

```
def process(x, y, verbose):
    verbose_manager = VerboseManager.instance()
    verbose_manager.start(2, verbose=verbose)
    # initialisation goes here
    verbose_manager.step("Step 1")
    # step 1 code goes here
    verbose_manager.step("Step 2")
    # step 2 code goes here
    verbose_manager.finish("Process")
```

Note that `VerboseManager` must be instantiated using `VerboseManager.instance()`; trying to instantiate it using `VerboseManager()` will throw a runtime error. The reason for this will be explained in subsection 3.1.

To start the process for `VerboseManager` instance `verbose_manager`, we use `verbose_manager.start(n_steps, verbose)`. This takes two arguments. `n_steps` is the number of steps the process involves - see in the example, this is 2 as the step method is called two times. Sadly, this must be calculated manually; this is unavoidable, as Python is not a compiled language[1]. The

[1] In a compiled language like C++, the number of steps can be calculated at compilation time; an interpreted language like Python has no way of 'seeing' where a step might be called in the future.

second parameter is `verbose`, which gives the verbosity level for the process. The levels are like so:

- Verbosity 0: no output is given. This is here to save processes from having to include a "`if verbosity>0; verbose_manager.start()`" block - if verbosity is 0, the manager is called and ignored entirely.
- Verbosity 1; no output is given during the process, but when a process finishes, the time taken for the whole process is printed, and the time taken for a process and all its subprocesses is returned as a variable when each process finishes.
- Verbosity 2; the features of verbosity 1, and additionally and additionally at each step a timing is given next to the progress bar with how long the previous step took; at the end of the process, a list of timings for each step is printed.
- Verbosity 3; the features of verbosity 1 and 2, and additionally a progress bar is given with progress percentage and a message for each step.

After starting, we can advance verbose printing for the process using the method `verbose_manager.step(message)`. This advances the process by one step, increasing the progress bar and also giving the `message` parameter as a message next to it[2]. This message should describe what the current step is doing.

When the process is done, verbose management can be completed using `verbose_manager.finish(process_name)`. This will end verbose management for the process (or subprocess). If it is a main process, time taken will be printed and returned as a variable. If it is a subprocess, time taken for the subprocess will be returned as a variable. `process_name` is, of course, a string with the name of the process; final timings will be printed in the format "[process_name] complete in [time] seconds.". Note that if n_steps was set incorrectly, at this point `VerboseManager` will throw a warning telling you how many steps it actually took; if your process takes a fixed number of steps, you could use this as a quick way of calculating n_steps.

At verbosity 3, the example given above would print the following at the end of the process:

```
[===============] 100% Complete
Process completed in [time] seconds.
Timings per step:
Initialisation: [time]
Step 1: [time]
Step 2: [time]
```

[2]I also like the message parameter as a way of making a process more self-explanatory and literate; when one is reading the code, the message can be treated like a comment on what the following block is doing.

3 Subprocess management

If a process involves sub-processes, like so:

```
def subprocess():
    # subprocess goes here

def process():
    # initialisation goes here
    # process code goes here
    subprocess()
    # more process code goes here
```

it is incredibly easy to add the subprocess to `VerboseManager`'s subprocess management. You simply do the following:

```
def subprocess():
    verbose_manager = VerboseManager.instance()
    verbose_manager.start(1, verbose=verbose)
    verbose_manager.step("Subprocess step")
    # subprocess goes here
    verbose_manager.finish("Subprocess")

def process():
    verbose_manager = VerboseManager.instance()
    verbose_manager.start(3, verbose=verbose)
    # initialisation goes here
    verbose_manager.step("Step 1")
    # step 1 code goes here
    subprocess()
    verbose_manager.step("Step 2")
    # step 2 code goes here
    verbose_manager.finish("Process")
```

that is, to add the subprocess we simply instantiate, start, step and finish the exact same way as we do in the main process. `VerboseManager` will recognise that a process is already being managed, and nest the steps of the subprocess into it.

The only caveat is that `n_steps` in the outermost (main) process must account for the steps contained in subprocesses. The `n_steps` in the subprocess is ignored entirely (and would only be used if one was directly calling the subprocess). This is for a reason; if we added the subprocess steps to the main process' management as we went, the progress bar would be incorrect before subprocesses get factored in. At step 1, it would say we were 50% complete, whereas in fact we have only done 1 out of 3 steps, so we are in fact 33% complete. This error would not be corrected until the subprocess starts, leading to

weird and inconsistent output. In fact, if a process had 2 steps and contained a subprocess with 3 steps, the progress bar would go from 50% to 40% (1/2 to 2/5) when the subprocess starts.

This example would output like so at the end of the process on verbosity 3:

```
[===============] 100% Complete
Process completed in [time] seconds.
Timings per step:
Initialisation: [time]
Step 1: [time]
|Subprocess step: [time]
Step 2: [time]
```

A vertical bar is used to denote the nesting level of subprocesses. As an example of this, if the subprocess had multiple steps including its own subprocess, and that sub-subprocess had its own subprocess, it would nest like so:

```
[===============] 100% Complete
Process completed in [time] seconds.
Timings per step:
Initialisation: [time]
Step 1: [time]
|Subprocess step 1: [time]
||Sub-subprocess step: [time]
|||Sub-sub-subprocess step: [time]
|Subprocess step 2: [time]
Step 2: [time]
```

We can also get timings for a whole subprocess as a return variable, that is `timings = verbose_manager.finish("Subprocess")`. This will return the time for the entire subprocess, regardless of how many steps or sub-subprocesses it goes into. These subprocess timings are stored simply as a stack; for each new subprocess, its start time is stored, then retrieved and popped when that subprocess finishes.

We will now go into the more technical details of how subprocesses are managed in `VerboseManager`.

3.1 Singleton class

`VerboseManager` uses a Singleton design pattern to be globally available to subprocesses. A Singleton pattern is an object where only one instance can exist at any one time, but that single instance has a global point of access[3]; it is often used for loggers. The Singleton design pattern in Python can be done simply like so:

```
class Singleton:
    _instance = None

    def __init__(self):
        raise RuntimeError("Singleton should not be instantiated directly.
                           " Use Singleton.instance().")

    @classmethod
    def instance(cls):
        """
        Instantiates a Singleton if one does not exist,
        and returns the existing one if it does exist.
        """
        if cls._instance is None:
            cls._instance = cls.__new__(cls)
            # initialisation goes here, for example
            # to initialise the attribute 'foo', use
            cls._instance.foo = bar
        return cls._instance
```

The `__init__` method is disabled entirely, and the class method `instance` is used instead. A class method is a method which operates on the entire concept of the class Singleton, rather than any specific instance on it. We give the class itself the attribute `_instance`, which will contain the single instance of Singleton that is allowed to exist. If one does not exist, one is created and initialised; note the code inside the `if` block is essentially what `__init__` does behind the scenes. If an instance does exist, it is returned.

This means that when a subprocess calls `VerboseManager.instance()`, it is given the exact same `VerboseManager` object as the one currently managing the main process. A step that takes place inside a subprocess is handled by the exact same instance that handles the main process' steps.

[3]The relevant pages from Design Patterns, explaining the specification of a Singleton, are available at https://archive.org/details/designpatternsel00gamm/page/127/mode/2up.

3.2 `start` and `finish` inside subprocesses

When a main process starts, `verbose_manager.start()` will change a bool `self._in_progress` from False to True. Likewise, `verbose_manager.finish()` will change it back to False when the main process ends. If `self._in_progress` already equals True and `verbose_manager.start()` is called again, `VerboseManager` will recognise that it has been called inside a subprocess. In this case, it will do the following:

```
self.subprocesses += 1
self.subprocess_start_times.append(time())
```

This accounts for the subprocess that has started, and also adds its start time to the list `self.subprocess_start_times`, which is treated as a stack. `self.subprocesses` records the 'nesting level'; i.e. how many subprocesses deep the process is.

When `verbose_manager.finish()` is called, it checks `self.subprocesses` to see if any subprocesses are running. If `self.subprocesses` equals 0, it finishes process management as the process that called it must be the main process. If `self.subprocesses` is greater than 0, it recognises that it has been called inside a subprocess, and accounts for it:

```
self.subprocesses -= 1
timings = time() - self.subprocess_start_times[-1]
self.subprocess_start_times.pop()
return timings
```

Treating `self.subprocess_start_times` as a stack ensures it gets the correct start time; `self.subprocess_start_times[-1]` will always be the start time of the most recently initialised subprocess, i.e. the most nested one. Subprocess time return can be useful for various other user outputs; e.g. for an iterative process, you can output the average time per iteration step using this feature.

Note that `VerboseManager` can only account for subprocesses that have a `verbose_manager.finish()` line; if a process is not printing final results when finished, you may have forgotten to finish a subprocess somewhere.