bprofile

Release 1.3

Dec 13, 2017

Contents

Chris Billington, Dec 13, 2017 *bprofile* is a wrapper around *cProfile*, *gprof2dot* and *graphviz*, providing a simple context manager for profiling sections of Python code and producing visual graphs of profiling results. It works on Windows and Unix.

View on PyPI | Get the source from BitBucket | Read the docs at readthedocs

CHAPTER 1

Installation

to install *bprofile*, run:

```
$ pip install bprofile
```

or to install from source:

```
$ python setup.py install
```

Note: *bprofile* requires graphviz to be installed. *bprofile* looks for a *graphviz* installation folder in `C:\Program Files` or `C:\Program Files (x86)` on Windows, and for *graphviz* executables in the `PATH` on Unix.

CHAPTER 2

Introduction

Every time I need to profile some Python code I go through the same steps: looking up *cProfile*'s docs, and then reading about *gprof2dot* and *graphviz*. And then it turns out the code I want to profile is a GUI callback or something, and I don't want to profile the whole program because it spends most of its time doing nothing.

cProfile certainly has this functionality, which I took one look at, and thought: *This should be a context manager, and when it exits, it should call gprof2dot and graphviz automatically so I don't have to remember their command line arguments, and so I don't accidentally print a .png to standard output and have to listen to all the ASCII beep characters.*

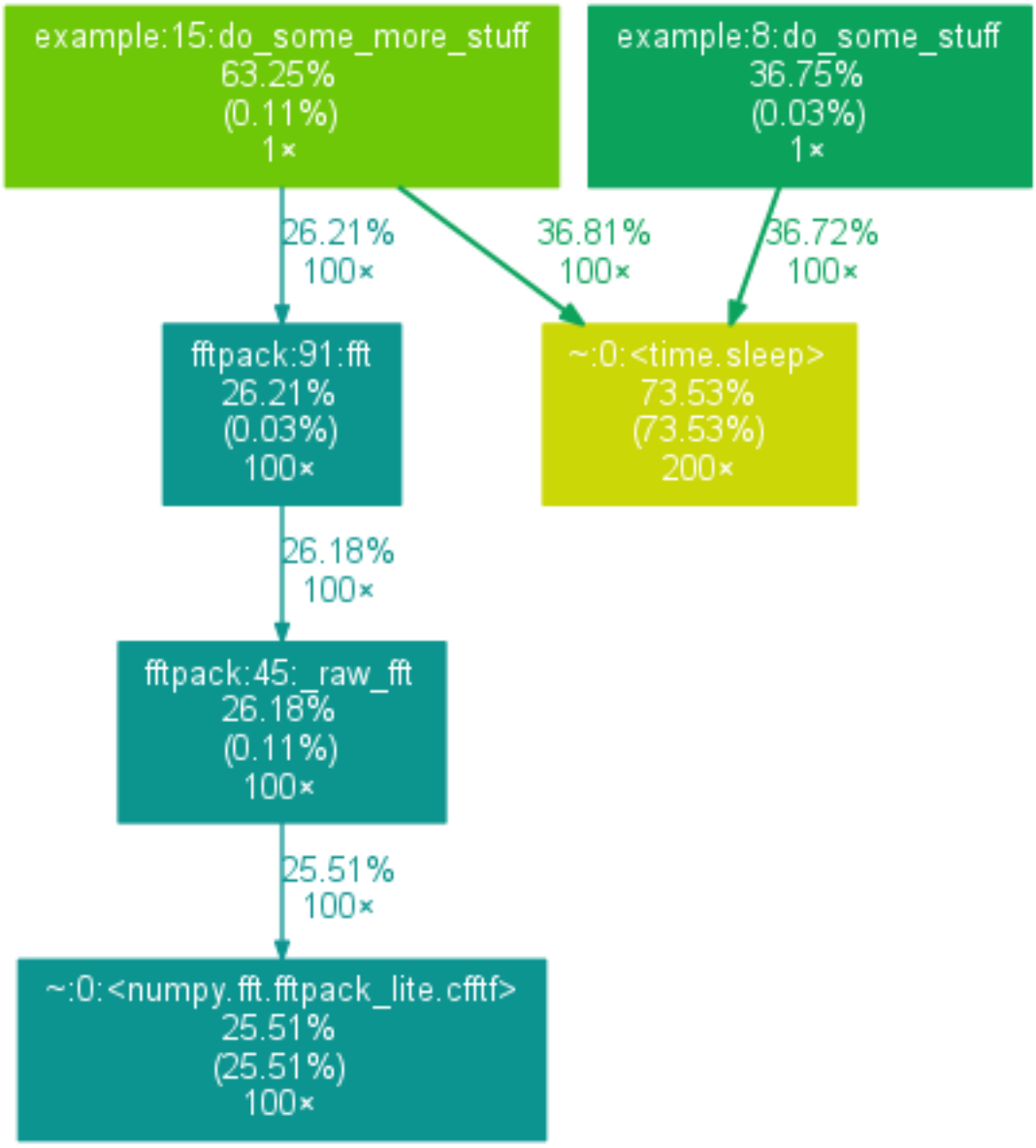
`BProfile` provides this functionality.

CHAPTER 3

Example usage

```
# example.py

import os
import time
import pylab as pl
from bprofile import BProfile

def do_some_stuff():
    for i in range(100):
        time.sleep(.01)

def do_some_stuff_that_wont_be_profiled():
    os.system('ping -c 5 google.com')

def do_some_more_stuff(n):
    x = pl.rand(100000)
    for i in range(100):
        time.sleep(.01)
        x = pl.fft(x)


profiler = BProfile('example.png')

with profiler:
    do_some_stuff()

do_some_stuff_that_wont_be_profiled()

with profiler:
    do_some_more_stuff(5)
```

The above outputs the following image `example.png` in the current working directory:



see *BProfile* for more information on usage.

CHAPTER 4

Class reference

class bprofile.**BProfile**(*output_path*, *threshold_percent=2.5*, *report_interval=5*, *enabled=True*)

A profiling context manager.

A context manager that after it exits, outputs a .png file of a graph made via cProfile, gprof2dot and graphviz. The context manager can be used multiple times, and if used repeatedly, regularly updates its output to include cumulative results.

An instance can also be used as a decorator, it will simply wrap calls to the decorated method in the profiling context.

Parameters

- **output_path** (*str*) – The name of the .png report file you would like to output. '.png' will be appended if not present.
- **threshold_percent** (*int or float*) – Nodes in which execution spends less than this percentage of the total profiled execution time will not be included in the output.
- **report_interval** (*int or float*) – The minimum time, in seconds, in between output file generation. If the context manager exits and it has not been at least this long since the last output was generated, output generation will be delayed until it has been. More profiling can run in the meantime. This is to decrease overhead on your program, (even though this overhead will only be incurred when no code is being profiled), while allowing you to have ongoing results of the profiling while your code is still running. If you only use the context manager once, then this argument has no effect. If you set it to zero, output will be produced after every exit of the context.
- **enabled** (*bool*) – Whether the profiler is enabled or not. Equivalent to calling *set_enabled()* with this argument after instantiation. Useful for enabling and disabling profiling with a global flag when you do not have easy access to the instance - for example when using as a decorator.

Notes

The profiler will return immediately after the context manager, and will generate its .png report in a separate thread. If the same context manager is used multiple times output will be generated at most every `report_interval` seconds (default: 5). The delay is to allow blocks to execute many times in between reports, rather than slowing your program down with generating graphs all the time. This means that if your profile block is running rapidly and repeatedly, a new report will be produced every `report_interval` seconds.

Pending reports will be generated at interpreter shutdown.

Note that even if `report_interval` is short, reporting will not interfere with the profiling results themselves, as a lock is acquired that will prevent profiled code from running at the same time as the report generation code. So the overhead produced by report generation does not affect the results of profiling - this overhead will only affect portions of your code that are not being profiled.

The lock is shared between instances, and so you can freely instantiate many *`BProfile`* instances to profile different parts of your code. Instances with the same `output_path` will share an underlying cProfile profiler, and so their reports will be combined. Profile objects are thread safe, so a single instance can be shared as well anywhere in your program.

Warning: Since only one profiler can be running at a time, two profiled pieces of code in different threads waiting on each other in any way will deadlock.

do_report()
Collect statistics and output a .png file of the profiling report.

Notes

This occurs automatically at a rate of `report_interval`, but one can call this method to report results sooner. The report will include results from all *`BProfile`* instances that have the same `output_path` and no more automatic reports (if further profiling is done) will be produced until after the minimum `report_interval` of those instances.

This method can be called at any time and is threadsafe. It is not advisable to call it during profiling however as this will incur overhead that will affect the profiling results. Only automatic reports are guaranteed to be generated when no profiling is taking place.

set_enabled(*enabled*)
Set whether profiling is enabled.

if enabled==True, all methods work as normal. Otherwise *`start()`*, *`stop()`*, and *`do_report()`* become dummy methods that do nothing. This is useful for having a global variable to turn profiling on or off, based on whether one is debugging or not, or to enable or disable profiling of different parts of code selectively.

If profiling is running when this method is called to disable it, the profiling will be stopped.

start()
Begin profiling.

stop()
Stop profiling.

Stop profiling and outptut a profiling report, if at least `report_interval` has elapsed since the last report. Otherwise output the report after a delay.

Does not preclude starting profiling again at a later time. Results are cumulative.

Python Module Index

b

Index