profile.doc   last updated 6/23/94 [by Guido]

PROFILER DOCUMENTATION and (mini) USER'S MANUAL

Written by James Roskind

The profiler was written after only programming in Python for 3 weeks. As a result, it is probably clumsy code, but I don't know for sure yet 'cause I'm a beginner :-). I did work hard to make the code run fast, so that profiling would be a reasonable thing to do. I tried not to repeat code fragments, but I'm sure I did some stuff in really awkward ways at times. Please send suggestions for improvements to: jar@infoseek.com. I won't promise *any* support. ...but I'd appreciate the feedback.

SECTION HEADING LIST:

REFERENCE MANUAL
  FUNCTION profile.run(string, filename_opt)
  CLASS Stats(filename, ...)
  METHOD strip_dirs()
  METHOD add(filename, ...)
  METHOD sort_stats(key, ...)
  METHOD reverse_order()
  METHOD print_stats(restriction, ...)
  METHOD print_callers(restrictions, ...)
  METHOD print_callees(restrictions, ...)
  METHOD ignore()
LIMITATIONS
CALIBRATION
EXTENSIONS: Deriving Better Profilers

INTRODUCTION

A "profiler" is a program that describes the run time performance of a program, providing a variety of statistics. This documentation describes the profiler functionality provided in the modules "profile" and "pstats." This profiler provides "deterministic profiling" of any Python programs. It also provides a series of report generation tools to allow users to rapidly examine the results of a profile operation.

HOW IS THIS profile DIFFERENT FROM THE OLD profile MODULE?

The big changes from standard profiling module are that you get more information, and you pay less CPU time. It's not a trade-off, it's a trade-up.

To be specific:

bugs removed: local stack frame is no longer molested, execution time is now charged to correct functions, ...

accuracy increased: profiler execution time is no longer charged to user's code, calibration for platform is supported, file reads are not done *by* profiler *during* profiling (and charged to user's code!), ...

speed increased: Overhead CPU cost was reduced by more than a factor of two (perhaps a factor of five), lightweight profiler module is all that must be loaded, and the report generating module (pstats) is not needed during profiling.

recursive functions support: cumulative times in recursive functions are correctly calculated; recursive entries are counted; ...

large growth in report generating UI: distinct profiles runs can be added together forming a comprehensive report; functions that import statistics take arbitrary lists of files; sorting criteria is now based on keywords (instead of 4 integer options); reports shows what functions were profiled as well as what profile file was referenced; output format has been improved, ...

INSTANT USERS MANUAL

This section is provided for users that "don't want to read the manual." It provides a very brief overview, and allows a user to rapidly perform profiling on an existing application.

To profile an application with a main entry point of "foo()", you would add the following to your module:

```
import profile
profile.run("foo()")
```

The above action would cause "foo()" to be run, and a series of informative lines (the profile) to be printed. The above approach is most useful when working with the interpreter. If you would like to save the results of a profile into a file for later examination, you can supply a file name as the second argument to the run() function:

```
import profile
profile.run("foo()", 'fooprof')
```

When you wish to review the profile, you should use the methods in the pstats module. Typically you would load the statistics data as follows:

```
import pstats
p = pstats.Stats('fooprof')
```

The class "Stats" (the above code just created an instance of this class) has a variety of methods for manipulating and printing the data that was just read into "p". When you ran profile.run() above, what was printed was the result of three method calls:

```
p.strip_dirs().sort_stats(-1).print_stats()
```

The first method removed the extraneous path from all the module names. The second method sorted all the entries according to the standard module/line/name string that is printed (this is to comply with the semantics of the old profiler). The third method printed out all the statistics. You might try the following sort calls:

```
p.sort_stats('name')
p.print_stats()
```

The first call will actually sort the list by function name, and the second call will print out the statistics. The following are some interesting calls to experiment with:

```
p.sort_stats('cumulative').print_stats(10)
```

This sorts the profile by cumulative time in a function, and then only prints the ten most significant lines. If you want to understand what algorithms are taking time, the above line is what you would use.

If you were looking to see what functions were looping a lot, and taking a lot of time, you would do:

```
p.sort_stats('time').print_stats(10)
```

to sort according to time spent within each function, and then print the statistics for the top ten functions.

You might also try:

```
p.sort_stats('file').print_stats('__init__')
```

This will sort all the statistics by file name, and then print out statistics for only the class init methods ('cause they are spelled with "__init__" in them). As one final example, you could try:

```
p.sort_stats('time', 'cum').print_stats(.5, 'init')
```

This line sorts stats with a primary key of time, and a secondary key of cumulative time, and then prints out some of the statistics. To be specific, the list is first culled down to 50% (re: .5) of its original size, then only lines containing "init" are maintained, and that sub-sub-list is printed.

If you wondered what functions called the above functions, you could now (p is still sorted according to the last criteria) do:

```
    p.print_callers(.5, 'init')
```

and you would get a list of callers for each of the listed functions.

If you want more functionality, you're going to have to read the manual (or guess) what the following functions do:

```
    p.print_callees()
    p.add('fooprof')
```

WHAT IS DETERMINISTIC PROFILING?

"Deterministic profiling" is meant to reflect the fact that all "function call", "function return", and "exception" events are monitored, and precise timings are made for the intervals between these events (during which time the user's code is executing). In contrast, "statistical profiling" (which is not done by this module) randomly samples the effective instruction pointer, and deduces where time is being spent. The latter technique traditionally involves less overhead (as the code does not need to be instrumented), but provides only relative indications of where time is being spent.

In Python, since there is an interpreter active during execution, the presence of instrumented code is not required to do deterministic profiling. Python automatically provides a hook (optional callback) for each event. In addition, the interpreted nature of Python tends to add so much overhead to execution, that deterministic profiling tends to only add small processing overhead, in typical applications. The result is that deterministic profiling is not that expensive, but yet provides extensive run time statistics about the execution of a Python program.

Call count statistics can be used to identify bugs in code (surprising counts), and to identify possible inline-expansion points (high call counts). Internal time statistics can be used to identify hot loops that should be carefully optimized. Cumulative time statistics should be used to identify high level errors in the selection of algorithms. Note that the unusual handling of cumulative times in this profiler allows statistics for recursive implementations of algorithms to be directly compared to iterative implementations.

REFERENCE MANUAL

The primary entry point for the profiler is the global function profile.run(). It is typically used to create any profile information. The reports are formatted and printed using methods for the class pstats.Stats. The following is a description of all of these standard entry points and functions. For a more in-depth view of some of the code, consider reading the later section on "Profiler Extensions," which includes discussion of how to derive "better" profilers from the classes presented, or reading the source code for these modules.

FUNCTION   profile.run(string, filename_opt)

This function takes a single argument that has can be passed to the "exec" statement, and an optional file name. In all cases this routine attempts to "exec" its first argument, and gather profiling statistics from the execution. If no file name is present, then this function automatically prints a simple profiling report, sorted by the standard name string (file/line/function-name) that is presented in each line. The following is a typical output from such a call:

cut here----

```
         main()
         2706 function calls (2004 primitive calls) in 4.504 CPU seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        2    0.006    0.003    0.953    0.477 pobject.py:75(save_objects)
     43/3    0.533    0.012    0.749    0.250 pobject.py:99(evaluate)
    ...
```

cut here----

The first line indicates that this profile was generated by the call: profile.run('main()'), and hence the exec'ed string is 'main()'. The second line indicates that 2706 calls were monitored. Of those calls, 2004 were "primitive." We define "primitive" to mean that the call was not induced via recursion. The next line: "Ordered by: standard name", indicates that the text string in the far right column was used to sort the output. The column headings include:

```
    "ncalls" for the number of calls,
    "tottime" for the total time spent in the given function
        (and excluding time made in calls to sub-functions),
    "percall" is the quotient of "tottime" divided by "ncalls"
    "cumtime" is the total time spent in this and all subfunctions
        (i.e., from invocation till exit). This figure is
        accurate *even* for recursive functions.
    "percall" is the quotient of "cumtime" divided by primitive
        calls
    "filename:lineno(function)" provides the respective data of
        each function
```

When there are two numbers in the first column (e.g.: 43/3), then the latter is the number of primitive calls, and the former is the actual number of calls. Note that when the function does not recurse, these two values are the same, and only the single figure is printed.

CLASS   Stats(filename, ...)

This class constructor creates an instance of a statistics object from a filename (or set of filenames). Stats objects are manipulated by methods, in order to print useful reports.

The file selected by the above constructor must have been created by the corresponding version of profile. To be specific, there is *NO* file compatibility guaranteed with future versions of this profiler, and there is no compatibility with files produced by other profilers (e.g., the standard system profiler).

If several files are provided, all the statistics for identical functions will be coalesced, so that an overall view of several processes can be considered in a single report. If additional files need to be combined with data in an existing Stats object, the add() method can be used.

METHOD strip_dirs()

This method for the Stats class removes all leading path information from file names. It is very useful in reducing the size of the printout to fit within (close to) 80 columns. This method modifies the object, and the striped information is lost. After performing a strip operation, the object is considered to have its entries in a "random" order, as it was just after object initialization and loading. If strip_dir() causes two function names to be indistinguishable (i.e., they are on the same line of the same filename, and have the same function name), then the statistics for these two entries are accumulated into a single entry.

METHOD add(filename, ...)

This methods of the Stats class accumulates additional profiling information into the current profiling object. Its arguments should refer to filenames created my the corresponding version of profile.run(). Statistics for identically named (re: file, line, name) functions are automatically accumulated into single function statistics.

METHOD sort_stats(key, ...)

This method modifies the Stats object by sorting it according to the supplied criteria. The argument is typically a string identifying the basis of a sort (example: "time" or "name").

When more than one key is provided, then additional keys are used as secondary criteria when the there is equality in all keys selected before them. For example, sort_stats('name', 'file') will sort all the entries according to their function name, and resolve all ties (identical function names) by sorting by file name.

Abbreviations can be used for any key names, as long as the abbreviation is unambiguous. The following are the keys currently defined:

Valid Arg	Meaning
"calls"	call count

"cumulative"	cumulative time
"file"	file name
"module"	file name
"pcalls"	primitive call count
"line"	line number
"name"	function name
"nfl"	name/file/line
"stdname"	standard name
"time"	internal time

Note that all sorts on statistics are in descending order (placing most time consuming items first), where as name, file, and line number searches are in ascending order (i.e., alphabetical). The subtle distinction between "nfl" and "stdname" is that the standard name is a sort of the name as printed, which means that the embedded line numbers get compared in an odd way. For example, lines 3, 20, and 40 would (if the file names were the same) appear in the string order "20" "3" and "40". In contrast, "nfl" does a numeric compare of the line numbers. In fact, sort_stats("nfl") is the same as sort_stats("name", "file", "line").

For compatibility with the standard profiler, the numeric argument -1, 0, 1, and 2 are permitted. They are interpreted as "stdname", "calls", "time", and "cumulative" respectively. If this old style format (numeric) is used, only one sort key (the numeric key) will be used, and additionally arguments will be silently ignored.

METHOD reverse_order()

This method for the Stats class reverses the ordering of the basic list within the object. This method is provided primarily for compatibility with the standard profiler. Its utility is questionable now that ascending vs descending order is properly selected based on the sort key of choice.

METHOD print_stats(restriction, ...)

This method for the Stats class prints out a report as described in the profile.run() definition.

The order of the printing is based on the last sort_stats() operation done on the object (subject to caveats in add() and strip_dirs()).

The arguments provided (if any) can be used to limit the list down to the significant entries. Initially, the list is taken to be the complete set of profiled functions. Each restriction is either an integer (to select a count of lines), or a decimal fraction between 0.0 and 1.0 inclusive (to select a percentage of lines), or a regular expression (to pattern match the standard name that is printed). If several restrictions are provided, then they are applied sequentially. For example:

```
    print_stats(.1, "foo:")
```

would first limit the printing to first 10% of list, and then only print functions that were part of filename ".*foo:". In contrast, the command:

```
    print_stats("foo:", .1)
```

would limit the list to all functions having file names ".*foo:", and then proceed to only print the first 10% of them.

METHOD print_callers(restrictions, ...)

This method for the Stats class prints a list of all functions that called each function in the profiled database. The ordering is identical to that provided by print_stats(), and the definition of the restricting argument is also identical. For convenience, a number is shown in parentheses after each caller to show how many times this specific call was made. A second non-parenthesized number is the cumulative time spent in the function at the right.

METHOD print_callees(restrictions, ...)

This method for the Stats class prints a list of all function that were called by the indicated function. Aside from this reversal of direction of calls (re: called vs was called by), the arguments and ordering are identical to the print_callers() method.

METHOD ignore()

This method of the Stats class is used to dispose of the value returned by earlier methods. All standard methods in this class return the instance that is being processed, so that the commands can be strung together. For example:

```
pstats.Stats('foofile').strip_dirs().sort_stats('cum').print_stats().ignore()
```

would perform all the indicated functions, but it would not return the final reference to the Stats instance.

LIMITATIONS

There are two fundamental limitations on this profiler. The first is that it relies on the Python interpreter to dispatch "call", "return", and "exception" events. Compiled C code does not get interpreted, and hence is "invisible" to the profiler. All time spent in C code (including builtin functions) will be charged to the Python function that was invoked the C code. IF the C code calls out to some native Python code, then those calls will be profiled properly.

The second limitation has to do with accuracy of timing information. There is a fundamental problem with deterministic profilers involving accuracy. The most obvious restriction is that the underlying "clock" is only ticking at a rate (typically) of about .001 seconds. Hence no measurements will be more accurate that that underlying clock. If enough measurements are taken, then the "error" will tend to average out. Unfortunately, removing this first error induces a second source of error...

The second problem is that it "takes a while" from when an event is dispatched until the profiler's call to get the time actually *gets* the state of the clock. Similarly, there is a certain lag when exiting the profiler event handler from the time that the clock's value was obtained (and then squirreled away), until the user's code is once again executing. As a result, functions that are called many times, or call many functions, will typically accumulate this error. The error that accumulates in this fashion is typically less than the accuracy of the clock (i.e., less than one clock tick), but it *can* accumulate and become very significant. This profiler provides a means of calibrating itself for a give platform so that this error can be probabilistically (i.e., on the average) removed. After the profiler is calibrated, it will be more accurate (in a least square sense), but it will sometimes produce negative numbers (when call counts are exceptionally low, and the gods of probability work against you :-).) Do *NOT* be alarmed by negative numbers in the profile. They should *only* appear if you have calibrated your profiler, and the results are actually better than without calibration.

CALIBRATION

The profiler class has a hard coded constant that is added to each event handling time to compensate for the overhead of calling the time function, and socking away the results. The following procedure can be used to obtain this constant for a given platform (see discussion in LIMITATIONS above).

```
import profile
pr = profile.Profile()
pr.calibrate(100)
pr.calibrate(100)
pr.calibrate(100)
```

The argument to calibrate() is the number of times to try to do the sample calls to get the CPU times. If your computer is *very* fast, you might have to do:

```
pr.calibrate(1000)
```

or even:

```
pr.calibrate(10000)
```

The object of this exercise is to get a fairly consistent result. When you have a consistent answer, you are ready to use that number in the source code. For a Sun Sparcstation 1000 running Solaris 2.3, the magical number is about .00053. If you have a choice, you are better off with a smaller constant, and your results will "less often" show up as negative in profile statistics.

The following shows how the trace_dispatch() method in the Profile class should be modified to install the calibration constant on a Sun Sparcstation 1000:

```
def trace_dispatch(self, frame, event, arg):
    t = self.timer()
```

```
        t = t[0] + t[1] - self.t - .00053 # Calibration constant

        if self.dispatch[event](frame,t):
            t = self.timer()
            self.t = t[0] + t[1]
        else:
            r = self.timer()
            self.t = r[0] + r[1] - t # put back unrecorded delta
        return
```

Note that if there is no calibration constant, then the line containing the callibration constant should simply say:

```
        t = t[0] + t[1] - self.t  # no calibration constant
```

You can also achieve the same results using a derived class (and the profiler will actually run equally fast!!), but the above method is the simplest to use. I could have made the profiler "self calibrating", but it would have made the initialization of the profiler class slower, and would have required some *very* fancy coding, or else the use of a variable where the constant .00053 was placed in the code shown. This is a ****VERY**** critical performance section, and there is no reason to use a variable lookup at this point, when a constant can be used.

EXTENSIONS: Deriving Better Profilers

The Profile class of profile was written so that derived classes could be developed to extend the profiler. Rather than describing all the details of such an effort, I'll just present the following two examples of derived classes that can be used to do profiling. If the reader is an avid Python programmer, then it should be possible to use these as a model and create similar (and perchance better) profile classes.

If all you want to do is change how the timer is called, or which timer function is used, then the basic class has an option for that in the constructor for the class. Consider passing the name of a function to call into the constructor:

```
    pr = profile.Profile(your_time_func)
```

The resulting profiler will call your time function instead of os.times(). The function should return either a single number, or a list of numbers (like what os.times() returns). If the function returns a single time number, or the list of returned numbers has length 2, then you will get an especially fast version of the dispatch routine.

Be warned that you *should* calibrate the profiler class for the timer function that you choose. For most machines, a timer that returns a lone integer value will provide the best results in terms of low overhead during profiling. (os.times is *pretty* bad, 'cause it returns a tuple of floating point values, so all arithmetic is floating point in the profiler!). If you want to be substitute a better timer in the cleanest fashion, you should derive a class, and simply put in the replacement dispatch method that better handles your timer call, along with the appropriate calibration constant :-).

```
cut here------------------------------------------------------------------
#*****************************************************************************
# OldProfile class documentation
#*****************************************************************************
#
# The following derived profiler simulates the old style profile, providing
# errant results on recursive functions. The reason for the usefulness of this
# profiler is that it runs faster (i.e., less overhead) than the old
# profiler.  It still creates all the caller stats, and is quite
# useful when there is *no* recursion in the user's code.  It is also
# a lot more accurate than the old profiler, as it does not charge all
# its overhead time to the user's code.
#*****************************************************************************
class OldProfile(Profile):
    def trace_dispatch_exception(self, frame, t):
        rt, rtt, rct, rfn, rframe, rcur = self.cur
        if rcur and not rframe is frame:
            return self.trace_dispatch_return(rframe, t)
        return 0

    def trace_dispatch_call(self, frame, t):
        fn = `frame.f_code`

        self.cur = (t, 0, 0, fn, frame, self.cur)
        if self.timings.has_key(fn):
            tt, ct, callers = self.timings[fn]
            self.timings[fn] = tt, ct, callers
```

```
        else:
            self.timings[fn] = 0, 0, {}
        return 1

    def trace_dispatch_return(self, frame, t):
        rt, rtt, rct, rfn, frame, rcur = self.cur
        rtt = rtt + t
        sft = rtt + rct

        pt, ptt, pct, pfn, pframe, pcur = rcur
        self.cur = pt, ptt+rt, pct+sft, pfn, pframe, pcur

        tt, ct, callers = self.timings[rfn]
        if callers.has_key(pfn):
            callers[pfn] = callers[pfn] + 1
        else:
            callers[pfn] = 1
        self.timings[rfn] = tt+rtt, ct + sft, callers

        return 1


    def snapshot_stats(self):
        self.stats = {}
        for func in self.timings.keys():
            tt, ct, callers = self.timings[func]
            nor_func = self.func_normalize(func)
            nor_callers = {}
            nc = 0
            for func_caller in callers.keys():
                nor_callers[self.func_normalize(func_caller)]=\
                      callers[func_caller]
                nc = nc + callers[func_caller]
            self.stats[nor_func] = nc, nc, tt, ct, nor_callers



#****************************************************************************
# HotProfile class documentation
#****************************************************************************
#
# This profiler is the fastest derived profile example.  It does not
# calculate caller-callee relationships, and does not calculate cumulative
# time under a function.  It only calculates time spent in a function, so
```

```
# it runs very quickly (re: very low overhead).  In truth, the basic
# profiler is so fast, that is probably not worth the savings to give
# up the data, but this class still provides a nice example.
#***************************************************************************
class HotProfile(Profile):
    def trace_dispatch_exception(self, frame, t):
        rt, rtt, rfn, rframe, rcur = self.cur
        if rcur and not rframe is frame:
            return self.trace_dispatch_return(rframe, t)
        return 0

    def trace_dispatch_call(self, frame, t):
        self.cur = (t, 0, frame, self.cur)
        return 1

    def trace_dispatch_return(self, frame, t):
        rt, rtt, frame, rcur = self.cur

        rfn = `frame.f_code`

        pt, ptt, pframe, pcur = rcur
        self.cur = pt, ptt+rt, pframe, pcur

        if self.timings.has_key(rfn):
            nc, tt = self.timings[rfn]
            self.timings[rfn] = nc + 1, rt + rtt + tt
        else:
            self.timings[rfn] =      1, rt + rtt

        return 1


    def snapshot_stats(self):
        self.stats = {}
        for func in self.timings.keys():
            nc, tt = self.timings[func]
            nor_func = self.func_normalize(func)
            self.stats[nor_func] = nc, nc, tt, 0, {}



cut here------------------------------------------------------------------
```