NAME

Benchmark - benchmark running times of Perl code

SYNOPSIS

```
   use Benchmark qw(:all) ;

   timethis ($count, "code");

   # Use Perl code in strings...
   timethese($count, {
'Name1' => '...code1...',
'Name2' => '...code2...',
   });

   # ... or use subroutine references.
   timethese($count, {
'Name1' => sub { ...code1... },
'Name2' => sub { ...code2... },
   });

   # cmpthese can be used both ways as well
   cmpthese($count, {
'Name1' => '...code1...',
'Name2' => '...code2...',
   });

   cmpthese($count, {
'Name1' => sub { ...code1... },
'Name2' => sub { ...code2... },
   });

   # ...or in two stages
   $results = timethese($count,
       {
    'Name1' => sub { ...code1... },
    'Name2' => sub { ...code2... },
       },
'none'
   );
   cmpthese( $results ) ;

   $t = timeit($count, '...other code...')
   print "$count loops of other code took:",timestr($t),"\n";

   $t = countit($time, '...other code...')
   $count = $t->iters ;
   print "$count loops of other code took:",timestr($t),"\n";

   # enable hires wallclock timing if possible
   use Benchmark ':hireswallclock';
```

DESCRIPTION

The Benchmark module encapsulates a number of routines to help you figure out how long it takes to execute some code.

timethis - run a chunk of code several times

timethese - run several chunks of code several times

cmpthese - print results of timethese as a comparison chart

timeit - run a chunk of code and see how long it goes

countit - see how many times a chunk of code runs in a given time

Methods

new

Returns the current time. Example:

```
use Benchmark;
$t0 = Benchmark->new;
# ... your code here ...
$t1 = Benchmark->new;
$td = timediff($t1, $t0);
print "the code took:",timestr($td),"\n";
```

debug

Enables or disable debugging by setting the `$Benchmark::Debug` flag:

```
Benchmark->debug(1);
$t = timeit(10, ' 5 ** $Global ');
Benchmark->debug(0);
```

iters

Returns the number of iterations.

Standard Exports

The following routines will be exported into your namespace if you use the Benchmark module:

timeit(COUNT, CODE)

Arguments: COUNT is the number of times to run the loop, and CODE is the code to run. CODE may be either a code reference or a string to be eval'd; either way it will be run in the caller's package.

Returns: a Benchmark object.

timethis (COUNT, CODE, [TITLE, [STYLE]])

Time COUNT iterations of CODE. CODE may be a string to eval or a code reference; either way the CODE will run in the caller's package. Results will be printed to STDOUT as TITLE followed by the times. TITLE defaults to "timethis COUNT" if none is provided. STYLE determines the format of the output, as described for timestr() below.

The COUNT can be zero or negative: this means the *minimum number of CPU seconds* to run. A zero signifies the default of 3 seconds. For example to run at least for 10 seconds:

```
timethis(-10, $code)
```

or to run two pieces of code tests for at least 3 seconds:

```
timethese(0, { test1 => '...', test2 => '...'})
```

CPU seconds is, in UNIX terms, the user time plus the system time of the process itself, as opposed to the real (wallclock) time and the time spent by the child processes. Less than 0.1 seconds is not accepted (-0.01 as the count, for example, will cause a fatal runtime exception).

Note that the CPU seconds is the **minimum** time: CPU scheduling and other operating system factors may complicate the attempt so that a little bit more time is spent. The benchmark output will, however, also tell the number of `$code` runs/second, which should be a more interesting number than the actually spent seconds.

Returns a Benchmark object.

timethese (COUNT, CODEHASHREF, [STYLE])

The CODEHASHREF is a reference to a hash containing names as keys and either a string to eval or a code reference for each value. For each (KEY, VALUE) pair in the CODEHASHREF, this routine will call

```
timethis(COUNT, VALUE, KEY, STYLE)
```

The routines are called in string comparison order of KEY.

The COUNT can be zero or negative, see timethis().

Returns a hash reference of Benchmark objects, keyed by name.

timediff (T1, T2)

Returns the difference between two Benchmark times as a Benchmark object suitable for passing to timestr().

timestr (TIMEDIFF, [STYLE, [FORMAT]])

Returns a string that formats the times in the TIMEDIFF object in the requested STYLE. TIMEDIFF is expected to be a Benchmark object similar to that returned by timediff().

STYLE can be any of 'all', 'none', 'noc', 'nop' or 'auto'. 'all' shows each of the 5 times available ('wallclock' time, user time, system time, user time of children, and system time of children). 'noc' shows all except the two children times. 'nop' shows only wallclock and the two children times. 'auto' (the default) will act as 'all' unless the children times are both zero, in which case it acts as 'noc'. 'none' prevents output.

FORMAT is the *printf(3)*-style format specifier (without the leading '%') to use to print the times. It defaults to '5.2f'.

Optional Exports

The following routines will be exported into your namespace if you specifically ask that they be imported:

clearcache (COUNT)

Clear the cached time for COUNT rounds of the null loop.

clearallcache ( )

Clear all cached times.

cmpthese (COUNT, CODEHASHREF, [STYLE])

cmpthese (RESULTSHASHREF, [STYLE])

Optionally calls timethese(), then outputs comparison chart. This:

```
    cmpthese( -1, { a => "++\$i", b => "\$i *= 2" } ) ;
```

outputs a chart like:

```
           Rate    b    a
    b 2831802/s   -- -61%
    a 7208959/s 155%   --
```

This chart is sorted from slowest to fastest, and shows the percent speed difference between each pair of tests.

`cmpthese` can also be passed the data structure that timethese() returns:

```
    $results = timethese( -1, { a => "++\$i", b => "\$i *= 2"
 } ) ;
    cmpthese( $results );
```

in case you want to see both sets of results. If the first argument is an unblessed hash reference, that is RESULTSHASHREF; otherwise that is COUNT.

Returns a reference to an ARRAY of rows, each row is an ARRAY of cells from the above chart, including labels. This:

```
    my $rows = cmpthese( -1, { a => '++$i', b => '$i *= 2' },
 "none" );
```

returns a data structure like:

```
    [
        [ '',       'Rate',   'b',    'a' ],
        [ 'b', '2885232/s',  '--', '-59%' ],
        [ 'a', '7099126/s', '146%',  '--' ],
    ]
```

NOTE: This result value differs from previous versions, which returned the `timethese()` result structure. If you want that, just use the two statement `timethese`...`cmpthese` idiom shown above.

Incidently, note the variance in the result values between the two examples; this is typical of benchmarking. If this were a real benchmark, you would probably want to run a lot more iterations.

countit(TIME, CODE)

Arguments: TIME is the minimum length of time to run CODE for, and CODE is the code to run. CODE may be either a code reference or a string to be eval'd; either way it will be run in the caller's package.

TIME is *not* negative. countit() will run the loop many times to calculate the speed of CODE before running it for TIME. The actual time run for will usually be greater than TIME due to system clock resolution, so it's best to look at the number of iterations divided by the times that you are concerned with, not just the iterations.

Returns: a Benchmark object.

disablecache ( )

Disable caching of timings for the null loop. This will force Benchmark to recalculate these timings for each new piece of code timed.

enablecache ( )

Enable caching of timings for the null loop. The time taken for COUNT rounds of the null loop will be calculated only once for each different COUNT used.

timesum (T1, T2)

Returns the sum of two Benchmark times as a Benchmark object suitable for passing to timestr().

:hireswallclock

If the Time::HiRes module has been installed, you can specify the special tag `:hireswallclock` for Benchmark (if Time::HiRes is not available, the tag will be silently ignored). This tag will cause the wallclock time to be measured in microseconds, instead of integer seconds. Note though that the speed computations are still conducted in CPU time, not wallclock time.

NOTES

The data is stored as a list of values from the time and times functions:

```
(
```

in seconds for the whole loop (not divided by the number of rounds).

The timing is done using time(3) and times(3).

Code is executed in the caller's package.

The time of the null loop (a loop with the same number of rounds but empty loop body) is subtracted from the time of the real loop.

The null loop times can be cached, the key being the number of rounds. The caching can be controlled using calls like these:

```
clearcache($key);
clearallcache();

disablecache();
enablecache();
```

Caching is off by default, as it can (usually slightly) decrease accuracy and does not usually noticably affect runtimes.

EXAMPLES

For example,

```
use Benchmark qw( cmpthese ) ;
$x = 3;
cmpthese( -5, {
    a => sub{$x*$x},
    b => sub{$x**2},
} );
```

outputs something like this:

```
Benchmark: running a, b, each for at least 5 CPU seconds...
       Rate    b    a
b 1559428/s   -- -62%
a 4152037/s 166%   --
```

while

```
use Benchmark qw( timethese cmpthese ) ;
$x = 3;
```

```
    $r = timethese( -5, {
        a => sub{$x*$x},
        b => sub{$x**2},
    } );
    cmpthese $r;
```

outputs something like this:

```
    Benchmark: running a, b, each for at least 5 CPU seconds...
             a: 10 wallclock secs ( 5.14 usr +  0.13 sys =  5.27 CPU) @
3835055.60/s (n=20210743)
             b:  5 wallclock secs ( 5.41 usr +  0.00 sys =  5.41 CPU) @
1574944.92/s (n=8520452)
               Rate    b    a
    b 1574945/s   -- -59%
    a 3835056/s 144%   --
```

INHERITANCE

Benchmark inherits from no other class, except of course for Exporter.

CAVEATS

Comparing eval'd strings with code references will give you inaccurate results: a code reference will show a slightly slower execution time than the equivalent eval'd string.

The real time timing is done using time(2) and the granularity is therefore only one second.

Short tests may produce negative figures because perl can appear to take longer to execute the empty loop than a short test; try:

```
    timethis(100,'1');
```

The system time of the null loop might be slightly more than the system time of the loop with the actual code and therefore the difference might end up being < 0.

SEE ALSO

Devel::DProf - a Perl code profiler

AUTHORS

Jarkko Hietaniemi <*jhi@iki.fi*>, Tim Bunce <*Tim.Bunce@ig.co.uk*>

MODIFICATION HISTORY

September 8th, 1994; by Tim Bunce.

March 28th, 1997; by Hugo van der Sanden: added support for code references and the already documented 'debug' method; revamped documentation.

April 04-07th, 1997: by Jarkko Hietaniemi, added the run-for-some-time functionality.

September, 1999; by Barrie Slaymaker: math fixes and accuracy and efficiency tweaks. Added cmpthese(). A result is now returned from timethese(). Exposed countit() (was runfor()).

December, 2001; by Nicholas Clark: make timestr() recognise the style 'none' and return an empty string. If cmpthese is calling timethese, make it pass the style in. (so that 'none' will suppress output). Make sub new dump its debugging output to STDERR, to be consistent with everything else. All bugs found while writing a regression test.

September, 2002; by Jarkko Hietaniemi: add ':hireswallclock' special tag.

February, 2004; by Chia-liang Kao: make cmpthese and timestr use time statistics for children instead of parent when the style is 'nop'.

November, 2007; by Christophe Grosjean: make cmpthese and timestr compute time consistently with style argument, default is 'all' not 'noc' any more.