`timeit` — Measure execution time of small code snippets

Source code: Lib/timeit.py

This module provides a simple way to time small bits of Python code. It has both a Command-Line Interface as well as a callable one. It avoids a number of common traps for measuring execution times. See also Tim Peters' introduction to the "Algorithms" chapter in the *Python Cookbook*, published by O'Reilly.

Basic Examples

The following example shows how the Command-Line Interface can be used to compare three different expressions:

```
$ python3 -m timeit '"-".join(str(n) for n in range(100))'
10000 loops, best of 5: 30.2 usec per loop
$ python3 -m timeit '"-".join([str(n) for n in range(100)])'
10000 loops, best of 5: 27.5 usec per loop
$ python3 -m timeit '"-".join(map(str, range(100)))'
10000 loops, best of 5: 23.2 usec per loop
```

This can be achieved from the Python Interface with:

```
>>> import timeit
>>> timeit.timeit('"-".join(str(n) for n in range(100))', number=10000)
0.3018611848820001
>>> timeit.timeit('"-".join([str(n) for n in range(100)])', number=10000)
0.2727368790656328
>>> timeit.timeit('"-".join(map(str, range(100)))', number=10000)
0.23702679807320237
```

A callable can also be passed from the Python Interface:

```
>>> timeit.timeit(lambda: "-".join(map(str, range(100))), number=10000)
0.19665591977536678
```

Note however that `timeit()` will automatically determine the number of repetitions only when the command-line interface is used. In the Examples section you can find more advanced examples.

Python Interface

The module defines three convenience functions and a public class:

`timeit.`**`timeit`**(*stmt='pass'*, *setup='pass'*, *timer=<default timer>*, *number=1000000*, *globals=None*)

Create a `Timer` instance with the given statement, *setup* code and *timer* function and run its `timeit()` method with *number* executions. The optional *globals* argument specifies a namespace in which to execute the code.

Changed in version 3.5: The optional *globals* parameter was added.

`timeit.`**`repeat`**(*stmt='pass'*, *setup='pass'*, *timer=<default timer>*, *repeat=5*, *number=1000000*, *globals=None*)

Create a `Timer` instance with the given statement, *setup* code and *timer* function and run its `repeat()` method with the given *repeat* count and *number* executions. The optional *globals* argument specifies a namespace in which to execute the code.

Changed in version 3.5: The optional *globals* parameter was added.

Changed in version 3.7: Default value of *repeat* changed from 3 to 5.

`timeit.`**`default_timer`**()

The default timer, which is always `time.perf_counter()`.

Changed in version 3.3: `time.perf_counter()` is now the default timer.

class `timeit.`**`Timer`**(*stmt='pass'*, *setup='pass'*, *timer=<timer function>*, *globals=None*)

Class for timing execution speed of small code snippets.

The constructor takes a statement to be timed, an additional statement used for setup, and a timer function. Both statements default to `'pass'`; the timer function is platform-dependent (see the module doc string). *stmt* and *setup* may also contain multiple statements separated by `;` or newlines, as long as they don't contain multi-line string literals. The statement will by default be executed within timeit's namespace; this behavior can be controlled by passing a namespace to *globals*.

To measure the execution time of the first statement, use the `timeit()` method. The `repeat()` and `autorange()` methods are convenience methods to call `timeit()` multiple times.

The execution time of *setup* is excluded from the overall timed execution run.

The *stmt* and *setup* parameters can also take objects that are callable without arguments. This will embed calls to them in a timer function that will then be executed by `timeit()`. Note that the timing overhead is a little larger in this case because of the extra function calls.

Changed in version 3.5: The optional *globals* parameter was added.

timeit(*number=1000000*)

Time *number* executions of the main statement. This executes the setup statement once, and then returns the time it takes to execute the main statement a number of times, measured in seconds as a float. The argument is the number of times through the loop, defaulting to one million. The main statement, the setup statement and the timer function to be used are passed to the constructor.

> **Note:** By default, `timeit()` temporarily turns off garbage collection during the timing. The advantage of this approach is that it makes independent timings more comparable. The disadvantage is that GC may be an important component of the performance of the function being measured. If so, GC can be re-enabled as the first statement in the *setup* string. For example:
>
> ```
> timeit.Timer('for i in range(10): oct(i)', 'gc.enable()').timeit()
> ```

autorange(*callback=None*)

Automatically determine how many times to call `timeit()`.

This is a convenience function that calls `timeit()` repeatedly so that the total time >= 0.2 second, returning the eventual (number of loops, time taken for that number of loops). It calls `timeit()` with increasing numbers from the sequence 1, 2, 5, 10, 20, 50, … until the time taken is at least 0.2 second.

If *callback* is given and is not `None`, it will be called after each trial with two arguments: `callback(number, time_taken)`.

New in version 3.6.

repeat(*repeat=5*, *number=1000000*)

Call `timeit()` a few times.

This is a convenience function that calls the `timeit()` repeatedly, returning a list of results. The first argument specifies how many times to call `timeit()`. The second argument specifies the *number* argument for `timeit()`.

> **Note:** It's tempting to calculate mean and standard deviation from the result vector and report these. However, this is not very useful. In a typical case, the lowest value gives a lower bound for how fast your machine can run the given code snippet; higher values in the result vector are typically not caused by variability in Python's speed, but by other processes interfering with your timing accuracy. So the `min()` of the result is probably the only number you should be interested in. After that, you should look at the entire vector and apply common sense rather than statistics.

Changed in version 3.7: Default value of *repeat* changed from 3 to 5.

print_exc(*file=None*)

Helper to print a traceback from the timed code.

Typical use:

```
t = Timer(...)       # outside the try/except
try:
    t.timeit(...)    # or t.repeat(...)
except Exception:
    t.print_exc()
```

The advantage over the standard traceback is that source lines in the compiled template will be displayed. The optional *file* argument directs where the traceback is sent; it defaults to `sys.stderr`.

Command-Line Interface

When called as a program from the command line, the following form is used:

```
python -m timeit [-n N] [-r N] [-u U] [-s S] [-h] [statement ...]
```

Where the following options are understood:

-n N, **--number**=N
how many times to execute 'statement'

-r N, **--repeat**=N
how many times to repeat the timer (default 5)

-s S, **--setup**=S
statement to be executed once initially (default `pass`)

-p, **--process**
measure process time, not wallclock time, using `time.process_time()` instead of `time.perf_counter()`, which is the default

New in version 3.3.

-u, **--unit**=U

specify a time unit for timer output; can select nsec, usec, msec, or sec

New in version 3.5.

-v, **--verbose**
print raw timing results; repeat for more digits precision

-h , --help

print a short usage message and exit

A multi-line statement may be given by specifying each line as a separate statement argument; indented lines are possible by enclosing an argument in quotes and using leading spaces. Multiple `-s` options are treated similarly.

If `-n` is not given, a suitable number of loops is calculated by trying successive powers of 10 until the total time is at least 0.2 seconds.

`default_timer()` measurements can be affected by other programs running on the same machine, so the best thing to do when accurate timing is necessary is to repeat the timing a few times and use the best time. The `-r` option is good for this; the default of 5 repetitions is probably enough in most cases. You can use `time.process_time()` to measure CPU time.

Note: There is a certain baseline overhead associated with executing a pass statement. The code here doesn't try to hide it, but you should be aware of it. The baseline overhead can be measured by invoking the program without arguments, and it might differ between Python versions.

Examples

It is possible to provide a setup statement that is executed only once at the beginning:

```
$ python -m timeit -s 'text = "sample string"; char = "g"'  'char in text'
5000000 loops, best of 5: 0.0877 usec per loop
$ python -m timeit -s 'text = "sample string"; char = "g"'  'text.find(char)'
1000000 loops, best of 5: 0.342 usec per loop
```

```
>>> import timeit
>>> timeit.timeit('char in text', setup='text = "sample string"; char = "g"')
0.41440500499993504
>>> timeit.timeit('text.find(char)', setup='text = "sample string"; char = "g"')
1.7246671520006203
```

The same can be done using the `Timer` class and its methods:

```
>>> import timeit
>>> t = timeit.Timer('char in text', setup='text = "sample string"; char = "g"')
>>> t.timeit()
0.3955516149999312
>>> t.repeat()
[0.40183617287970225, 0.37027556854118704, 0.38344867356679524, 0.37125959708466
```

The following examples show how to time expressions that contain multiple lines. Here we compare the cost of using `hasattr()` vs. `try`/`except` to test for missing and present object attributes:

```
$ python -m timeit 'try:' '  str.__bool__' 'except AttributeError:' '  pass'
20000 loops, best of 5: 15.7 usec per loop
$ python -m timeit 'if hasattr(str, "__bool__"): pass'
50000 loops, best of 5: 4.26 usec per loop

$ python -m timeit 'try:' '  int.__bool__' 'except AttributeError:' '  pass'
200000 loops, best of 5: 1.43 usec per loop
$ python -m timeit 'if hasattr(int, "__bool__"): pass'
100000 loops, best of 5: 2.23 usec per loop
```

```
>>> import timeit
>>> # attribute is missing
>>> s = """\
... try:
...     str.__bool__
... except AttributeError:
...     pass
... """
>>> timeit.timeit(stmt=s, number=100000)
0.9138244460009446
>>> s = "if hasattr(str, '__bool__'): pass"
>>> timeit.timeit(stmt=s, number=100000)
0.5829014980008651
>>>
>>> # attribute is present
>>> s = """\
... try:
...     int.__bool__
... except AttributeError:
...     pass
... """
>>> timeit.timeit(stmt=s, number=100000)
0.04215312199994514
>>> s = "if hasattr(int, '__bool__'): pass"
>>> timeit.timeit(stmt=s, number=100000)
0.08588060699912603
```

To give the `timeit` module access to functions you define, you can pass a *setup* parameter which contains an import statement:

```
def test():
    """Stupid test function"""
    L = [i for i in range(100)]

if __name__ == '__main__':
    import timeit
    print(timeit.timeit("test()", setup="from __main__ import test"))
```

Another option is to pass `globals()` to the *globals* parameter, which will cause the code to be executed within your current global namespace. This can be more convenient than individually specifying imports:

```
def f(x):
    return x**2
def g(x):
    return x**4
def h(x):
    return x**8

import timeit
print(timeit.timeit('[func(42) for func in (f,g,h)]', globals=globals()))
```