timeit — Time the execution of small bits of Python code.

Purpose: Time the execution of small bits of Python code.

The `timeit` module provides a simple interface for determining the execution time of small bits of Python code. It uses a platform-specific time function to provide the most accurate time calculation possible and reduces the impact of start-up or shutdown costs on the time calculation by executing the code repeatedly.

Module Contents

`timeit` defines a single public class, `Timer`. The constructor for `Timer` takes a statement to be timed and a "setup" statement (used to initialize variables, for example). The Python statements should be strings and can include embedded newlines.

The `timeit()` method runs the setup statement one time, then executes the primary statement repeatedly and returns the amount of time that passes. The argument to `timeit()` controls how many times to run the statement; the default is 1,000,000.

Basic Example

To illustrate how the various arguments to `Timer` are used, here is a simple example that prints an identifying value when each statement is executed.

```
# timeit_example.py

import timeit

# using setitem
t = timeit.Timer("print('main statement')", "print('setup')")

print('TIMEIT:')
print(t.timeit(2))

print('REPEAT:')
print(t.repeat(3, 2))
```

When run, the output shows the results of the repeated calls to `print()`.

```
$ python3 timeit_example.py

TIMEIT:
setup
main statement
main statement
1.8429999999944324e-06
REPEAT:
setup
main statement
main statement
setup
main statement
main statement
setup
main statement
main statement
[1.4149999999976681e-06, 1.005999999997842e-06,
1.0179999999984646e-06]
```

`timeit()` runs the setup statement one time, then calls the main statement `count` times. It returns a single floating point value representing the cumulative amount of time spent running the main statement.

When `repeat()` is used, it calls `timeit()` several times (3 in this case) and all of the responses are returned in a list.

Storing Values in a Dictionary

This more complex example compares the amount of time it takes to populate a dictionary with a large number of values using a variety of methods. First, a few constants are needed to configure the `Timer`. The `setup_statement` variable initializes a list of tuples containing strings and integers that will be used by the main statements to build dictionaries using the strings as keys and storing the integers as the associated values.

```
# A few constants
range_size = 1000
count = 1000
setup_statement = ';'.join([
    "l = [(str(x), x) for x in range(1000)]",
    "d = {}",
])
```

A utility function, `show_results()`, is defined to print the results in a useful format. The `timeit()` method returns the amount of time it takes to execute the statement repeatedly. The output of `show_results()` converts that into the amount of time it takes per iteration, and then further reduces the value to the average amount of time it takes to store one item in the dictionary.

```
def show_results(result):
    "Print microseconds per pass and per item."
    global count, range_size
    per_pass = 1000000 * (result / count)
    print('{:6.2f} usec/pass'.format(per_pass), end=' ')
    per_item = per_pass / range_size
    print('{:6.2f} usec/item'.format(per_item))


print("{} items".format(range_size))
print("{} iterations".format(count))
print()
```

To establish a baseline, the first configuration tested uses `__setitem__()`. All of the other variations avoid overwriting values already in the dictionary, so this simple version should be the fastest.

The first argument to `Timer` is a multi-line string, with white space preserved to ensure that it parses correctly when run. The second argument is a constant established to initialize the list of values and the dictionary.

```
# Using __setitem__ without checking for existing values first
print('__setitem__:', end=' ')
t = timeit.Timer(
    textwrap.dedent(
        """
        for s, i in l:
            d[s] = i
        """),
    setup_statement,
)
show_results(t.timeit(number=count))
```

The next variation uses `setdefault()` to ensure that values already in the dictionary are not overwritten.

```
# Using setdefault
print('setdefault :', end=' ')
t = timeit.Timer(
    textwrap.dedent(
        """
        for s, i in l:
            d.setdefault(s, i)
        """),
    setup_statement,
)
show_results(t.timeit(number=count))
```

This method adds the value only if a `KeyError` exception is raised when looking for the existing value.

```
# Using exceptions
```

```
# Using exceptions
print('KeyError   :', end=' ')
t = timeit.Timer(
    textwrap.dedent(
        """
        for s, i in l:
            try:
                existing = d[s]
            except KeyError:
                d[s] = i
        """),
    setup_statement,
)
show_results(t.timeit(number=count))
```

And the last method uses "in" to determine if a dictionary has a particular key.

```
# Using "in"
print('"not in"   :', end=' ')
t = timeit.Timer(
    textwrap.dedent(
        """
        for s, i in l:
            if s not in d:
                d[s] = i
        """),
    setup_statement,
)
show_results(t.timeit(number=count))
```

When run, the script produces the following output.

```
$ python3 timeit_dictionary.py

1000 items
1000 iterations

__setitem__:  62.47 usec/pass   0.06 usec/item
setdefault : 122.70 usec/pass   0.12 usec/item
KeyError   :  60.78 usec/pass   0.06 usec/item
"not in"   :  55.79 usec/pass   0.06 usec/item
```

Those times are for a MacMini, and will vary depending on what hardware is used and what other programs are running on the system. Experiment with the `range_size` and `count` variables, since different combinations will produce different results.

From the Command Line

In addition to the programmatic interface, `timeit` provides a command line interface for testing modules without instrumentation.

To run the module, use the `-m` option to the Python interpreter to find the module and treat it as the main program:

```
$ python3 -m timeit
```

For example, to get help:

```
$ python3 -m timeit -h

Tool for measuring execution time of small code snippets.

This module 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.

...
```

The `statement` argument works a little differently on the command line than the argument to `Timer`. Instead of using one long string, pass each line of the instructions as a separate command line argument. To indent lines (such as inside a loop), embed spaces in the string by enclosing it in quotes.

```
$ python3 -m timeit -s \
"d={}" \
"for i in range(1000):" \
"  d[str(i)] = i"

1000 loops, best of 5: 332 usec per loop
```

It is also possible to define a function with more complex code, then call the function from the command line.

```
# timeit_setitem.py


def test_setitem(range_size=1000):
    l = [(str(x), x) for x in range(range_size)]
    d = {}
    for s, i in l:
        d[s] = i
```

To run the test, pass in code that imports the modules and runs the test function.

```
$ python3 -m timeit \
"import timeit_setitem; timeit_setitem.test_setitem()"

1000 loops, best of 5: 376 usec per loop
```

See also

- Standard library documentation for timeit
- `profile` – The `profile` module is also useful for performance analysis.
- Monotonic Clocks – Discussion of the monotonic clock from the `time` module.

profile and pstats — Performance Analysis

tabnanny — Indentation validator

Quick Links

Module Contents
Basic Example
Storing Values in a Dictionary
From the Command Line

This page was last updated 2018-12-09.

Navigation

profile and pstats — Performance Analysis
tabnanny — Indentation validator

The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?

This Site

Module Index
Index

Other Writing

Blog
The Python Standard Library By Example